Daniel Sanchez

iOS-sovelluksen testaustyökalut

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

14.4.2018

Metropolia

Tekijä Otsikko Sivumäärä Aika	Daniel Sanchez iOS-sovelluksen testaustyökalut 40 sivua 14.4.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	mediatekniikka
Suuntautumisvaihtoehto	digitaalinen media
Ohjaaja	lehtori Olli Alm

Insinöörityön tarkoituksena oli perehtyä iOS-sovelluskehityksen perusteisiin ja kehitysympäristöön ja tutustua olemassa oleviin testaustyökaluihin. Työn tavoitteena oli kehittää iOS-käyttöjärjestelmälle mobiilisovellus, jota käytetään testauksen kohteena.

Insinöörityössä perehdyttiin mobiilisovelluksen testaukseen ja mobiililaitteen erikoispiirteisiin, jotka asettavat testaamiselle erityishaasteita. Lisäksi tarkasteltiin sovelluksen testaamista simulaattorilla tai oikealla laitteella ja niiden välisiä eroja, manuaalisten ja automatisoitujen testausten piirteitä ja niiden käytettävyyttä sovelluksen testaamisessa sekä muita mahdollisia tapoja testata sovellusta käyttämällä muun muassa prototyyppejä tai alfa- tai beetajakelua.

Työssä selvisi, että mobiilisovellusten testaamista varten on olemassa monipuolisesti erilaisia työkaluja, joita voi käyttää sovelluksen kehityksessä. Tarjolla olevien työkalujen lisäksi testauksessa kannattaa huomioida, milloin manuaalinen testaus on lopulta järkevämpää kuin automatisoitu testaus. Alfa- ja beeta-testausta kannattaa hyödyntää varhaisessa vaiheessa, jotta käyttäjiltä saadaan varhaista palautetta sovelluksesta ja sen toiminnoista.

Syntynyt iOS-mobiilisovellus noutaa Helsingin kaupungin reittioppaan avoimesta rajapinnasta reittitietoa ja esittää saadun tiedon sovelluksen käyttöliittymässä. Sovellukselle laadittiin yksikkö-, integraatio- ja käyttöliittymätestejä testauskirjastoilla, joihin työssä oli perehdytty. Testauskirjastojen välillä on eroja, jotka vaikuttavat testien kirjoittamiseen ja niiden luettavuuteen. Hyödyntämällä monipuolisesti tarjolla olevia testauskirjastoja voidaan parantaa sovelluksen laatua testauksessa.

Avainsanat	iOS, mobiilisovellukset, testaus, testausmenetelmät

Author Title Number of Pages Date	Daniel Sanchez The testing tools for an iOS application 40 pages 14 April 2018
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructor	Olli Alm, Senior Lecturer

The purpose of this thesis was to get familiar with the basics of iOS-development, development environment and to get acquainted with existing testing tools and to develop an iOS-application from scratch. The resulting mobile application would be used as a target to perform testing on.

During the thesis we get acquainted with the basics of mobile software testing and its unique qualities that set boundaries for mobile testing. In addition, we inspected the differences between testing with an actual device and with a simulator that emulates the behavior of an actual device. This thesis includes comparison between manual and automated testing and the usability of each method in mobile testing. Finally, we inspected other possible methods for testing by using prototypes and crowd testing.

During the thesis, it was discovered that there are a lot of different tools and libraries for mobile application testing that can be used during the development process. Even though there are plenty of tools available, one should consider whether manual testing is more beneficial than automated testing. Also crowd testing should be taken advantage of early on, as the development team will receive early feedback from the users of the application.

The application that was developed during this thesis fetches data from the Reittiopas API and displays it to the user. Different kinds of tests were written for this application such as unit, integration and functional user interface tests. There are differences between the testing libraries that affect how the tests will be written and therefore affect readability. By taking advantage of the existing testing libraries, we can build better quality software.

Keywords	iOS, mobile applications, testing, testing methods

Sisällys

Lyhenteet

1 Johdanto 1

2 Ohjelmistotestaus 2

2.1 Testaus osana ketterää ohjelmistokehitystä 3
2.2 Mobiilisovellusten testauksesta 6
2.3 Testaustavat 7
2.4 Testauspyramidi ja ohjelmistotestauksen tasot 10
2.5 Käyttöliittymäsuunnittelu ja prototyypit 13
2.6 Sovelluksen rajoitettu jakelu 14
2.7 Pilvitestaustyökalut 15
2.8 Jatkuva integraatio 16

3 iOS-käyttöjärjestelmä ja -kehitysympäristö 18

3.1 iOS-käyttöjärjestelmä 18
3.2 Kehitysympäristö 19
3.3 iOS-sovelluksen arkkitehtuuri 20
3.4 Tunnetuimmat alustakohtaiset testikirjastot 22

4 Sovelluksen kehitys ja testaus 25

4.1 Yleistä 25
4.2 Sovelluksen rakenne ja toteutus 27
4.3 Reittiopas-rajapinnan käyttö sovelluksessa 30
4.4 Sovelluksen testaus 30
4.5 Jatkuvan integraation käyttöönotto 34

5 Yhteenveto 36

Lähteet 37

Lyhenteet

BDD	Behaviour-driven development. Käyttäytymislähtöinen kehitys.
CI	Continuous integration. Jatkuva integraatio.
IDE	Integrated development environment. Integroitu editori.
SDK	Software development kit. Sovelluskirjasto.
TDD	Test-driven development. Testivetoinen kehitys.

1 Johdanto

Insinöörityön ensisijaisena tavoitteena on kehittää iOS-käyttöjärjestelmälle mobiilisovellus ja perehtyä sovelluksen testaamiseen tarjolla oleviin työkaluihin. Toisena tavoitteena on omaksua iOS-sovelluskehityksen perusteet ja tutustua iOS-kehittäjän ohjelmointiympäristöön, Xcode-editoriin, Swift-ohjelmointikieleen ja testaukseen tarkoitettuihin kirjastoihin. Työssä kehitetään iOS-sovellus, joka hakee Helsingin kaupungin Reittioppaan avoimesta rajapinnasta reittitietoa ja esittää rajapinnasta saadun sisällön sovelluksessa. Sovellusta testataan testaukseen painottuvilla avoimen lähdekoodin kirjastoilla, joihin työssä perehdytään.

Insinöörityössä keskitytään ohjelmistotestaukseen ja sovelluskehitykseen iOS-ympäristössä, minkä vuoksi työn keskipiste on iOS-mobiilisovellusten testaamisessa eikä tarkoituksena ole läpikäydä ohjelmistotestausta kaikessa laajuudessaan. Työssä esitellään erilaisia lähestymistapoja testaukseen: manuaalisesta testauksesta automatisoituun. Manuaalisessa ja automatisoidussa testauksessa käytetään työssä esiteltäviä ohjelmistotestauksen tasoja. Sen jälkeen käydään läpi vaihtoehtoisia tapoja testata sovellusta, esituotantovaiheen prototyypistä jo kehityksessä olevan sovelluksen rajoitettuun alfa- ja beeta-jakeluun sekä käsitellään lyhyesti jatkuva integraatio osana testausta. Työstä on rajattu pois ohjelmistotestauksen perusteita, kuten perinteinen testausprosessi, lasilaatikko ja musta laatikko -testausmenetelmät. Rajaus on tehty työn aihealueen rajoittamisen vuoksi, eivätkä nämä aihealueet kasvattaisi merkittävästi insinöörityön arvoa.

Luvussa 3 käsitellään iOS-alustan perusteet, iOS-sovelluksissa tyypillinen MVC-arkkitehtuurimalli sekä yleisimmät testaukseen liittyvät kirjastot.

Työssä kehitetty mobiilisovellus on toteutettu Xcode-ohjelmointiympäristössä, ja sovellus toimii pelkästään puhelimissa eikä tabletteja varten laadittu erillistä käyttöliittymää. Luvussa 4 esitetään sovelluksen rakenne, käytettävä rajapinta, sovelluksen riippuvuuksien hallinta ja esitellään muutama esimerkki testi sovelluksesta havainnollistamisen vuoksi. Lopulta käydään lyhyesti läpi jatkuva integraatio osana sovellusta nojautumalla CircleCI-palvelun ominaisuuksiin.

2 Ohjelmistotestaus

Ohjelmistotestaus on merkittävä osa ohjelmistotuotantoa, ja sen aikana työskennellään toteutettavan ohjelmiston eteen, jotta siitä tulee toivotun kaltainen ja se täyttää sille asetetut vaatimukset. Ohjelmistotestauksen aikana tarkastellaan, mitä on saatu aikaiseksi ja tunnistetaan ne kohdat, joissa ohjelmisto ei täytä sille asetettuja vaatimuksia. [Kasurinen 2013: s. 16.]

Ohjelmiston verifiointi ja validointi ovat tärkeä osa ohjelmistotestausta: niiden avulla varmistutaan oikean ohjelmiston rakentamisesta oikealla tavalla. Verifioinnin tarkoituksena on varmistaa, että ohjelmisto rakennetaan oikealla tavalla. Ohjelmiston pitää täyttää sille asetetut vaatimukset: sen pitää toimia luotettavasti, toipua erilaisista virhetilanteista ongelmitta eikä sen tulisi sisältää vikoja. Validoinnin tarkoituksena on varmistaa ohjelmiston oikeasti täyttävän asiakkaan ja loppukäyttäjän tarpeet ja vaatimukset sekä varmentaa ohjelmiston vaatimusmäärittelyn olevan puutteeton. [Laboon 2016: s. 12.]

Ohjelmiston virheettömyyden osoittaminen on toisaalta mahdotonta, mutta on mahdollista todistaa ohjelmiston sisältävän vikoja, koska ohjelmistossa voi olla rajaton määrä testitapauksia sen syötteiden, suorituspolkujen ja lopputulemien vuoksi. Tämän takia ohjelmiston testauksessa tulisi panostaa rajattuun testaamiseen optimaalisen resurssien käytön takia ja olisi hyvä muistaa, että hyviin testituloksiin ei voida täysin luottaa, koska ohjelmisto voi sisältää löytämättömiä vikoja. [Haikala & Mikkonen 2011: s. 205; Myers 2011: s. 8.]

On kuitenkin hyvä tiedostaa, että uusissa ja pitkäikäisissäkin ohjelmistoissa esiintyy erilaisia vikoja eikä kaikkia vikoja välttämättä ikinä löydetä. Uusissa ohjelmistoissa arvioidaan olevan yksi vika muutamaa kymmentä riviä kohden, ja pitkäikäisissä ohjelmistoissa arvioidaan olevan yksi vika tuhatta riviä kohden. Järjestelmässä saattaa olla huomaamattomia vikoja, eikä niitä välttämättä havaita pitkäikäisenkään ohjelmiston elinkaaren aikana. Huomaamattomat viat eivät välttämättä aiheuta ohjelmistossa häiriötilanteita, koska viat voivat korjautua ohjelmiston myöhemmässä suoritusvaiheessa jonkin tilanteen seurauksena. [Haikala & Mikkonen 2011: s. 205–206]

2.1 Testaus osana ketterää ohjelmistokehitystä

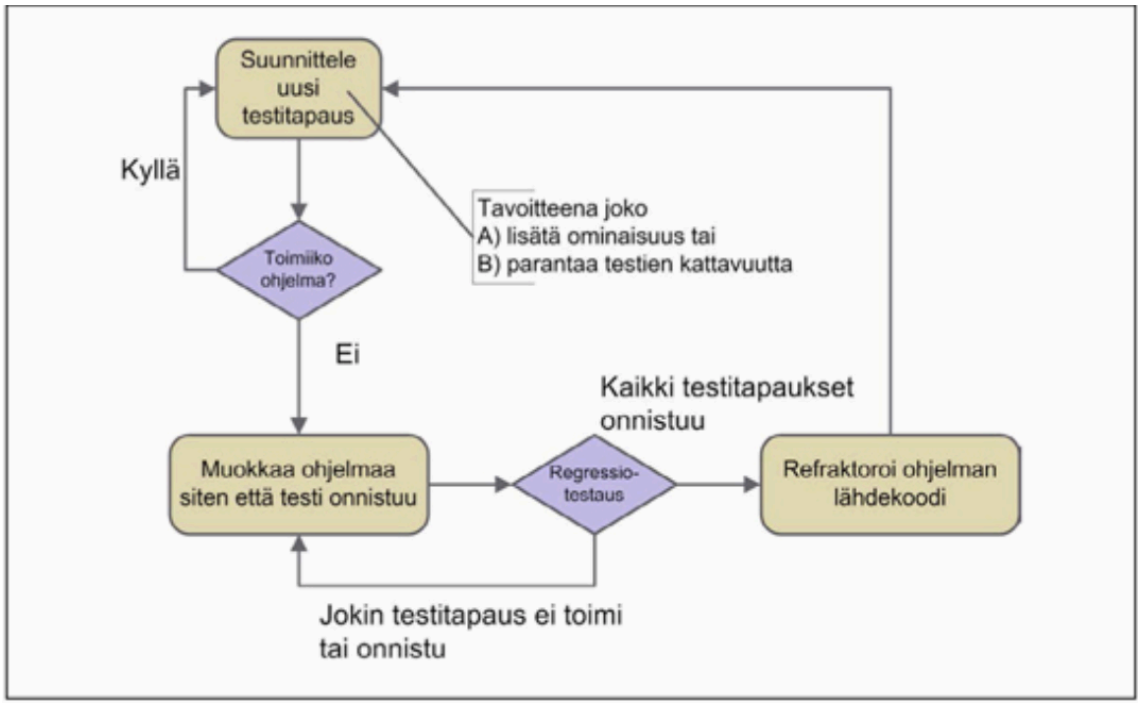
Nykypäivänä ketterien menetelmien käyttö ohjelmistokehityksessä parantaa lopputuotteen laatua, koska osa ketterien menetelmien keskeisistä periaatteista on sidoksissa testaamiseen. Ketterissä kehitystiimeissä ohjelmistokehitys on usein testivetoista, koska sitä pidetään hyvänä tapana kehittää ohjelmistoa ja se pakottaa varhaisessa vaiheessa miettimään ohjelmiston rakennetta sekä sen avulla voidaan välttyä virheiltä ohjelmistossa. Ketterät menetelmät ovat luonteeltaan iteratiivisia ja inkrementaalisia. Niiden soveltaminen testauksen piiriin käytännössä tarkoittaa, että sovelluksen jokainen inkrementaatio on testattu kehitysiteraation päättyessä. [Crispin 2009: s. 5, 8, 9.]

Ketterä kehitystiimi usein koostuu eri alojen ammattilaisista, jotka tekevät yhteistyötä sovelluksen eteen. Tiimiin voi kuulua markkinoinnin ammattilaisia, ohjelmistokehittäjiä, testaajia ja analyytikkoja. Joka tapauksessa tiimin osaaminen on laaja-alaista, ja kaikki ottavat kantaa sovelluksen kehitykseen, koska tarkoituksena on rakentaa mahdollisimman laadukas lopputulos. Jäsenien työtoimenkuva saattaa kuitenkin olla laajempi eikä rajoitu pelkästään yhteen aikaisemmin mainittuun rooliin. Ylipäätään tiimin jäsenien olisi hyvä pystyä liikkumaan kehityksen aikana muille osa-alueille tarpeen vaatiessa ja olla valmiita omaksumaan uusia toimintatapoja. [Tarlinder 2016: s. 401, 432; Crispin 2009: s. 6.]

Esimerkiksi sovelluskehittäjien tulisi ottaa vastuu kirjoittamastaan koodista eikä odottaa erillisen testaajan varmistavan koodin toimivuutta – vaikka kehitystiimissä olisi erillinen testaaja. Sovelluskehittäjä siis osallistuu ohjelmiston laadun ylläpitämiseen ja sen verifiointiin kirjoittamalla automatisoituja testejä kirjoittamalleen lähdekoodille, joka varmentaa ohjelmiston osion toimivan halutulla tavalla. Käytännössä sovelluskehittäjä toimii samanaikaisesti sovelluksen testaajana.

Sovelluskehittäjät voivat kirjoittaa testitapaukset koodilleen halutessaan etu- tai jälkikäteen, mutta tärkeintä on varmistaa kirjoittamansa koodin toimivan eikä pelkästään sen kääntyvän. Kehittäjät voivat kirjoittaa muun muassa automatisoituja yksikkö- ja integraatiotestejä ja käyttää jatkuvaa integraatiota (CI, continuous integration) testien suorittamiseen erillisellä palvelimella, joka ei ole sidoksissa sovelluskehittäjän ympäristöön. [Tarlinder 2016: s. 432, 443.]

Etukäteen kirjoitettavat testit yhdistetään testivetoiseen kehitykseen (TDD, Test Driven Development), joka on ohjelmistokehityksessä toimintamalli, jonka kehittäjä tai tiimi voi omaksua käytettäväksi. Testivetoisen kehityksen (kuva 1) periaatteena on aina kirjoittaa toiminnallisuutta testaava testi ensimmäisenä, minkä jälkeen kirjoitetaan tarvittava koodi, joka toteuttaa testattavan toiminnallisuuden. Käytännössä kehittäjä tekee useita lyhyitä iteraatioita, joiden aikana hän kirjoittaa pieniä osia toiminnallisuudesta. [Kasurinen 2013: s. 210; Tarlinder 2016: s. 443.]

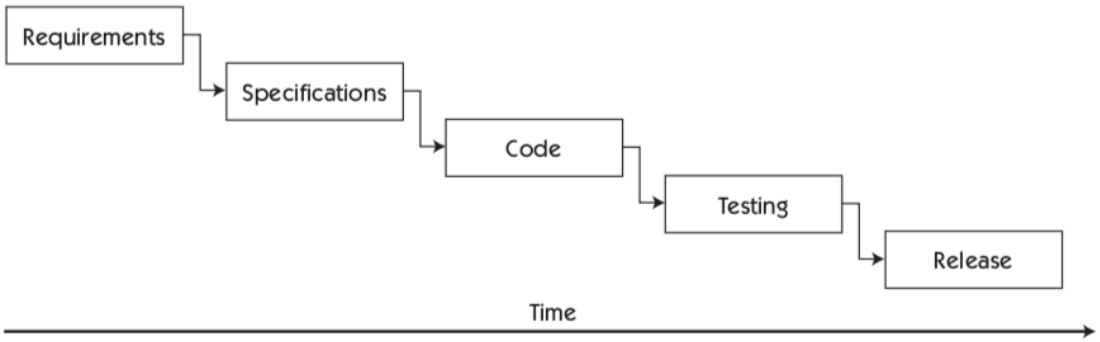


Kuva 1. Testivetoisen kehityksen vaiheet [Kasurinen 2013: s. 211]

Testivetoinen kehitys noudattaa prosessia, jonka kuva 1 havainnollistaa. Jokaisen iteraation alussa kirjoitetaan uusi testitapaus, jonka on tarkoituksena testata sovelluksen uutta ominaisuutta tai parantaa testikattavuutta. Kehittäjä toteuttaa varsinaisen toiminnallisuuden tai muutoksen, jota alkuvaiheessa kirjoitettu testi testaa. Heti kun toteutus täyttää testin asettamat vaatimukset, voidaan siirtyä ohjelman lähdekoodin refaktorointiin, jossa lähdekoodia parannetaan tai siistitään. Refaktoroinnin jälkeen voidaan siirtyä seuraavaan iteraation, ja iterointia jatketaan niin kauan, kunnes haluttu toiminnallisuus on vaatimusten mukaisesti toteutettu ja testattu. [Kasurinen 2013: s. 210.]

Testivetoisen kehityksen haittapuoliksi mielletään kuitenkin yksikkötestien suuri määrä, joka on lopulta riittämätön tapa testata sovelluksen kokonaisuutta tai ylipäätään sen kriittisiä osia. Toisena ongelmana voidaan pitää puutteellisesti laadittuja testejä ja niiden riittämätöntä tarkkuutta, joka antaa valheellisen kuvan sovelluksen toimivuudesta. Yksikkötestit eivät ole tarpeeksi luotettava tapa testata järjestelmää kokonaisuutena, mikä saattaa myöhemmässä vaiheessa vaatia laajoja muutoksia lähdekoodiin. [Kasurinen 2013: s. 212.]

Perinteisten ja ketterien menetelmien välillä on eroja ohjelmiston testauksen näkökulmasta. Perinteisessä vaiheittaisessa projektimallissa, kuten vesiputousmallissa testaaminen usein tehdään loppuvaiheessa toteutuksen ja julkaisun välimaastossa. Kuvan 2 diagrammi antaa ymmärtää, että kaikille työvaiheille on allokoitu yhtä paljon aikaa, mutta todellisuudessa myöhäisimmille työvaiheille usein jää vähemmän aikaa.



Kuva 2. Vesiputousmalli [Crispin 2009: s. 13].

Scrum-malli on yksi tunnetuimmista ketteristä kehityksen prosessimalleista, ja se on yleisesti käytetty pienemmissä organisaatioissa ja projekteissa, joissa kehitettävän tuotteen toiminnallisuudet viimeistellään tuotteen kehityksen aikana. Sen vuoksi se soveltuu erityisesti pelien ja mobiilisovelluksien kehitystä varten. Scrum-mallissa korostuu ketterien mallien mukaisesti yksilöiden välinen kommunikointi ja asiakasyhteistyö ja ohjelmiston toiminnollisuuksia suositaan enemmän kuin kattavaa dokumentointia. [Kasurinen 2013: s. 43–44.]

Ketterissä menetelmissä, kuten Scrum-mallissa, tuotekehitys koostuu useista muutaman viikon tai kuukausien pituisista iteraatioista, eli sprinteistä ja Scrum-prosessimalli on luonteeltaan inkrementaalista. Jokaisen iteraation aikana sovellusta kehitetään eteenpäin pienissä sykleissä, joissa lisätään tai muutetaan sovelluksen toiminnallisuutta. Käytännössä jokainen muutos tai uusi ominaisuus testataan, ennen kuin se todetaan valmiiksi. Toimintatavan seurauksena sovelluksen testauksen ei koskaan pitäisi laahata perässä, koska sovellukseen kohdistuneet muutokset merkitään valmiiksi vasta siinä vaiheessa, kun testaus on suoritettu loppuun. Jokaisen iteraation päättyessä sovelluksesta pitäisi olla testattu ja toimiva julkaisukelpoinen versio. [Crispin 2009: s. 12–14.]

2.2 Mobiilisovellusten testauksesta

Mobiilisovellusten testaaminen eroaa paljon perinteisten pöytäkone- tai verkkosovellusten testaamisesta useasta eri syystä, jotka ovat sidoksissa käyttökokemukseen tai sovelluksen tekniseen puoleen. Syitä ovat esimerkiksi käyttäjien korkeat odotukset, sovelluksen päivitystiheys, laitekannan laaja pirstaleisuus, verkkoyhteyden suorituskyky ja luotettavuus sekä laitteiden erilaiset ominaisuudet. Itse sovelluksen testaaminen ei välttämättä ole kaikista haastavin osuus, vaan sen toimivuuden varmistaminen eri laitteilla, käyttöjärjestelmäversioilla ja vielä erilaisissa olosuhteissa. [Knott 2015: s. 2–3; Myers 2011: s. 213–215.]

Erityisesti mobiilisovelluksissa, käyttöliittymän ja käyttökokemuksen pitää olla korkeatasoista, koska käyttäjillä on korkeammat odotukset mobiilisovellukselta kuin muiden alustojen sovelluksilta. Tilastojen mukaan valtaosa käyttäjistä poistaa sovelluksen ensimmäisen käyttökerran jälkeen, mikä yleensä johtuu huonosta käyttökokemuksesta, pitkistä latausajoista tai sovelluksen kaatumisesta ensimmäisellä käyttökerralla. Muita syitä sovelluksen poistamiselle ovat pakollinen sisäänkirjautuminen tai sovelluksen hidas käynnistyminen. [Knott 2015: s. 2–3.]

Mobiililaitteiden ja käyttöjärjestelmien pirstaleisuus on yksi mobiilisovellusten testaamisen haasteista. Sovellus saattaa käyttäytyä eri tavalla eri laitteissa, mikä voi johtua käyttöjärjestelmän versiosta tai laitevalmistajan erillisistä muutoksista käyttöjärjestelmään. Pelkästään erilaisten laitteiden ja käyttöjärjestelmäversioiden määrä on suuri, ja siksi kombinaatioiden riittävä testaaminen edellyttää automatisoitua testausta osana testausstrategiaa. [Cynthia 2014]

Verkkoyhteyden suorituskyvyn ja luotettavuuden testaaminen on suositeltavaa, koska sovelluksen loppukäyttäjän yhteys saattaa vaihdella hitaasta yhteydestä nopeampaan yhteyteen. Loppukäyttäjien erilaisten verkkoyhteyksien vuoksi sovellusta tulisi testata erilaisissa verkko-olosuhteissa, jotta voidaan varmistua sovelluksen käyttäytymisestä vaihtelevissa olosuhteissa. Verkkoyhteyden testaamisessa on tärkeätä huomioida, että sovelluksen käyttäjällä ei välttämättä ole verkkoyhteyttä lainkaan. [Arsene 2016]

Muita huomionarvoisia mobiilitestaamisen kohteita on sovelluksen kesketystilanteiden, tietoturvan ja lokalisoinnin testaus [Arsene 2016; Hechtel 2016a].

Keskeytystilanteiden testaamisessa käynnissä olevan sovelluksen toiminta keskeytyy sovelluksen ulkopuolisen tekijän seurauksena, esimerkiksi tekstiviestin saapumisen tai hälytyskellon soimisen. Keskeytystilanteessa käynnissä oleva sovellus pysähtyy väliaikaisesti, ja keskeytyksen päättyessä sovelluksen pitäisi jatkaa toimintaa tavalliseen tapaan siitä, mihin käyttäjä jäi. [Francino]

Sharkleyn mukaan lokalisointitestauksen tarkoituksena on varmistaa, että sovellus on lokalisoitu tietyille käyttäjäsegmenteille kohderyhmän vaatimusten mukaisesti – joko kulttuurin, kielen tai maan perusteella. Glezos tarkentaa, että lokalisoinnin kohteena on käyttöliittymäkomponenteissa esiintyvä sisältö, kuten sovelluksen kuvat ja tekstisisältö, jossa erityistä huomiota tulee kiinnittää kielioppiin sekä aika- ja valuuttaformaattiin [Sharkley; Glezos].

2.3 Testaustavat

Mobiilisovelluksen testaamisessa käytetään joko simulaattoria tai oikeaa fyysistä laitetta, joista molemmilla on omat vahvuutensa ja heikkoutensa.

Simulaattorin käytön etuna on sen ilmaisuus, nopeus ja suoraviivainen käyttöönotto. Välttämättä ei tarvita useita oikeita laitteita, mutta simulaattoreilla on merkittäviä heikkouksia, jotka liittyvät vahvasti laitteen käyttöön tai laitteen sensoreiden tuottamaan dataan. Simulaattorit eivät palauta todellista sensoridataa, minkä vuoksi niissä ajettavat sovellukset saattavat toimia odottamattomalla tavalla. Muina heikkouksina pidetään niiden kyvyttömyyttä tarjota todellista käyttöympäristöä ja käyttökokemusta.

Oikean laitteen käytöllä sen sijaan on huomattava määrä etuja verrattuna simulaattoreihin: laitteen nopeus, luotettavuus ja aito testausympäristö. Aidolla testausympäristöllä tarkoitetaan sitä, että laite voi käyttää monipuolisesti sen sisältämiä ominaisuuksia ja palauttaa todellista sensoridataa. Haittapuolena on laitteen hinta ja ylläpito, joka voi aiheuttaa ylimääräisiä kuluja. [Hechtel 2016b]

Kattavaa testaamista varten suositellaan käytettäväksi kaksitasoista (two-tier) testaamista, jossa testaamiseen käytetään simulaattoreiden lisäksi oikeita laitteita, jotka ovat markkinoiden uusimpia tai suosituimpia laitteita. Testauksen aikana oikealla laitteella testaaminen antaa realistisemman kuvan sovelluksen toiminnasta, koska simulaattorit eivät välttämättä tuota todellista lopputulosta tai käyttökokemusta. [Cynthia 2014]

Manuaalinen testaus

Manuaalisella testauksella tarkoitetaan sovelluksen testaamista käsin oikealla laitteella tai pilvipalvelussa, jossa varmennetaan sovelluksen toimivan odotetulla tavalla. Manuaalisen testauksen aloittaminen ei edellytä olemassa olevaa automatisointia tai testaamiseen soveltuvan kehitysympäristön pystyttämistä ja ylläpitoa, minkä vuoksi manuaalinen testaus on helppo aloittaa. Se saattaa olla joidenkin sovelluksien kohdalla edullisempi ratkaisu kuin automatisoitu lähestymistapa. Suoritettavan testauksen laajuuteen voi vaikuttaa olemassa olevien fyysisten testilaitteiden lukumäärä. Testilaitteiden hankinta ja ylläpito voi aiheuttaa ylimääräisiä kuluja - erityisesti ympäristöissä, joissa fyysinen pääsy laitteisiin on rajattu. [Mobile Testing: Manual Vs. Automation 2016]

Yleisesti manuaalisen testauksena vahvuutena ja heikkoutena voidaan pitää ihmistestaajan roolia testauksen aikana. Ihmisen kyky havaita poikkeamia sovelluksen käyttäytymisessä ja mahdollisuutta poiketa testisuunnitelmasta tarpeen vaatiessa ovat hyviä puolia, joiden avulla on mahdollista paikantaa sovelluksen toimintaan vaikuttavia epäkohtia. Huonona puolena on luonnollisesti ihmisen vaihteleva havainnointikyky eri olosuhteissa, minkä myötä osa sovelluksen virheistä ei välttämättä löydy tai osa niistä jää huomioimatta. Manuaalisen testauksen lopputulokseen vaikuttavat testaajan asenne ja olosuhteet: kiire, väsymys ja huolimattomuus voivat aiheuttaa tilanteen, jossa virheet jäävät löytämättä. [Mobile Testing: Manual Vs. Automation 2016; Crispin 2009: s. 259]

Ohjelmiston kompleksisuuden kasvaessa myös testitapauksen lukumäärä kasvaa, minkä seurauksena manuaaliseen testaukseen kuluu entistä enemmän resursseja. Pelkkään manuaaliseen testaukseen luottavassa tiimissä saattaa syntyä tilanne, jossa testaamisen allokoidut resurssit eivät tule riittämään. Testaamiselle allokoitujen resurssien loppuessa, testaaja ei kykene suoriutumaan ohjelmiston manuaalisesta regressiotestauksesta, esiintyvien virheiden selvittelystä ja uusien toimintojen kattavasta testauksesta. Resurssien loppuessa muu kehitystiimi saattaa ottaa osaa testaamiseen, mikä kuluttaa resursseja muualta, kuten uusien toimintojen toteuttamiselta. Tämän seurauksena ohjelmiston tekninen velka kasvaa ja kehitystiimi turhautuu. [Crispin 2009: s. 258]

Manuaalinen testaus on kuitenkin suhteellisen kallista ja virhealtista pitkäikäisissä projekteissa verrattuna luotettavaan automaatiokehykseen, jonka avulla testitapaukset suoritetaan luotettavasti ja täsmälleen samalla tavalla kuin edellisellä testauskierroksella. [Mobile Testing: Manual Vs. Automation 2016]

Automatisoitu testaus

Automatisoidussa testauksessa on kyse prosessista, jossa ohjelmistoa testataan automaattisesti sitä varten kehitetyillä automaatiotyövälineillä, jotka suorittavat ennalta laadittuja testitapauksia ja varmistavat niiden lopputuloksen odotetuksi. Yleisesti ajatellaan, että kattavan automatisoinnin seurauksena on usein laadukkaampi lopputuote.

Automatisointi ja automaattisesti suoritettavat testit luovat ympäristön, jossa testejä voidaan suorittaa usein nopeasti ja kustannustehokkaasti ja niiden olemassaolo kasvattaa sovelluksen arvoa sen elinkaaren ajan. Sen lisäksi niiden olemassa olo helpottavaa kehitystiimien päivittäistä työskentelyä ja niiden olemassa olo on edellytys ketteriä malleja noudattavilla tiimeille, koska ohjelmiston tulisi olla aina julkaisukuntoinen. [Mobile Testing: Manual Vs. Automation 2016; Crispin 2009: s. 258.]

Automaattiset testit suoritetaan ohjelmistoon kohdistuneen muutoksen yhteydessä, ja olemassa olevalle toiminnallisuudelle laaditun testin tulee mennä läpi ongelmitta niin kauan, kunnes toiminnallisuutta muutetaan. Mikäli ohjelmistoon tehty muutos on ollut tarkoituksenmukainen, sitä testaava testi tulisi päivittää vastaamaan uutta toiminnallisuutta.

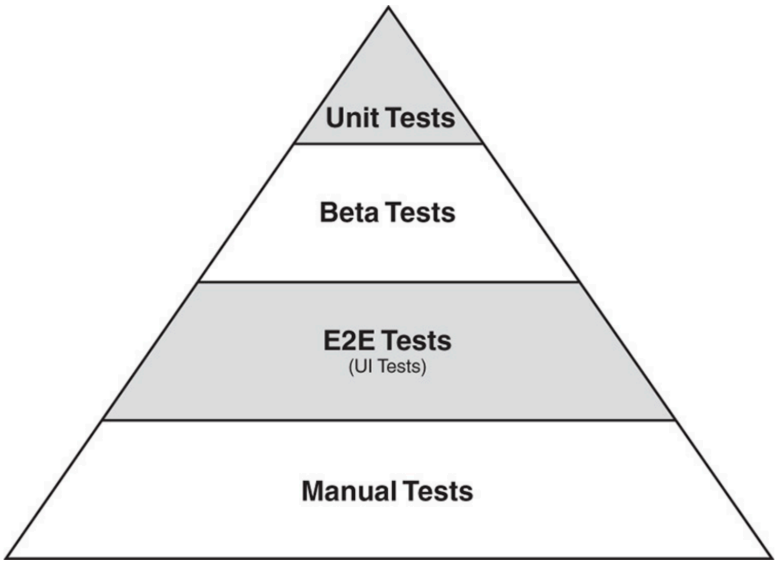
Ohjelmiston ylläpitäminen julkaisukuntoisena onnistuu automatisoitujen testien luomalla palautesilmukalla. Sen tarkoituksena on tarjota palautetta kehitystiimille, mikä puolestaan mahdollistaa nopean reagoinnin ohjelmiston epäkohtiin. Kehitystiimi saa palautesilmukan avulla tarkkaa ja tuoretta tietoa ohjelmiston hajoamisen ajankohdasta, aiheuttajasta ja syystä. Palautesilmukka edesauttaa virheen paikantamisessa ja sen korjaamisessa, koska virheen aiheuttanut muutos on vielä kehittäjän tai tiimin tuoreessa muistissa, minkä seurauksena virheenselvittelyyn ei kulu yhtä paljon aikaa kuin myöhemmässä ajankohdassa. [Crispin 2009: s. 258–262.]

Automaation tarkoituksena ei ole kuitenkaan poistaa manuaalista ja rutiininomaista testaustyötä täysin vaan täydentää sitä. Kun poistetaan osa manuaalisen testauksen aiheuttamasta kuormasta, se vapauttaa testaajan muihin tuottavampiin työtehtäviin. Testaaja voi esimerkiksi käyttää aikansa tehokkaammin löytääkseen uusia testitapauksia tai syventyäkseen paremmin ohjelmiston toimintaan. [Cynthia 2009: s. 259; Kasurinen 2013: s. 112–113.]

2.4 Testauspyramidi ja ohjelmistotestauksen tasot

Testauspyramidin jokainen osa kuvastaa tehtävää testauksen määrää tietyllä ohjelmistotestauksen tasolla. Knott esittää teoksessaan [2015: s. 108] oman version alkuperäisestä testauspyramidista, joka on laadittu sopivammaksi mobiilitestauksen piiriin. Senhetkiset automaatiotyökalut eivät ole riittävän kypsät kattavaan testauksen automatisointiin, jonka vuoksi manuaalinen testaus ja beeta-testaus on otettu osaksi hänen esittelemäänsä testauspyramidia. Knott perustelee manuaalisen testaamisen määrää epäkypsillä automaatiotyökaluilla, mutta toteaa manuaalisen testaamisen vähenevän tulevaisuudessa, kun työkalut kehittyvät tarpeeksi.

Pyramidi koostuu manuaalisesta ja automatisoidusta testauksesta. Kuvassa 3 valkoisella taustavärillä olevat pyramidin osat kuuluvat manuaalisen testauksen piiriin ja harmaat osiot automatisoidun testauksen piiriin. [Knott 2015: s. 105–108.]



Kuva 3. Mobiilitestauspyramidi [Knott 2015: s. 108].

Yksikkötestien (unit tests) tarkoituksena on testata ohjelmistoa pienessä mittakaavassa keskittymällä ohjelmiston yksittäisten moduulien, funktioiden tai objektien testaamiseen. Testattavan kohteen ulkoiset tekijät, kuten ulkoiset järjestelmät tai resurssit, poistetaan yksikkötesteistä, mikä mahdollistaa yksikkötestien ajamisen nopeasti ja luotettavasti missä tahansa ympäristössä. [Kaczanowski 2013: s. 13–14.]

Yksikkötesteissä ulkoiset järjestelmät, kuten ohjelmiston tietokanta, konfiguraatiotiedostot tai ulkopuoliset verkkopalvelut, korvataan erilaisilla testikopioilla, mikä mahdollistaa niiden suorittamisen nopeasti ja luotettavasti. Testikopioiden avulla simuloidaan vuorovaikutusta ulkopuolisten järjestelmien kanssa, jotta voidaan palauttaa ulkopuolisten järjestelmien sijaiskomponenteista haluttuja vastauksia tai virhetilanteita. [Kaczanowski 2013: s. 13–14; Kasurinen 2013: s. 80.]

Yksikkötestit voidaan yleisesti jakaa kahteen eri kategoriaan riippuen testin tyypistä: tilapohjaiseen (state-based) tai interaktiopohjaiseen (interaction-based). Tilapohjainen yksikkötesti varmentaa, että kohde palauttaa odotetun arvon tai sen tila muuttuu halutuksi. Interaktiopohjainen yksikkötesti varmentaa, että testikopioiden tiettyjä metodeja kutsutaan odotetulla tavalla. [Kolodiy]

Kolodiy listaa hyviä yksikkötestauksen piirteitä olevan helposti laadittavuus, nopeus, luettavuus ja luotettavuus. Wackler [2015] lisää, että yksikkötestien pitää olla luotettavia, koska niiden tarkoituksena on testata ohjelmiston pieniä osia ja antaa tarkkaa palautetta

ohjelmiston toimivuudesta. Pienten osien testaamisen sivuvaikutuksena testiä varten kirjoitettavan koodin määrä on myös pieni, mikä vähentää testissä esiintyvien virheiden määrää. [Kolodiy; Wackler 2015]

Integrointitestaus (integration testing) on usein seuraava työvaihe yksikkötestauksen jälkeen. Integrointitestauksessa yksittäiset komponentit integroidaan osaksi isompaa kokonaisuutta, minkä takia integrointitestit ovat laajempia testejä kuin yksikkötestit. Testien tarkoituksena on varmistaa, että yksittäiset komponentit toimivat yhdessä eikä uusien komponenttien integrointi aiheuta ohjelmistossa ongelmia. [Kasurinen 2013: s. 84–85.]

Järjestelmätestauksen (system testing) tarkoituksena on nimensä mukaisesti testata koko järjestelmää kerralla, kun kaikki järjestelmän komponentit on integroitu yhdeksi toimivaksi kokonaisuudeksi. Järjestelmätestausta tehdään yksikkö- ja integrointitestauksen jälkeen testiympäristössä, eikä tässä testauksen työvaiheessa enää käytetä integrointitestauksessa käytettyjä sijaiskomponentteja. Järjestelmätestauksen aikana varmistutaan järjestelmän oikeellisuudesta ja toimivuudesta, ja samalla etsitään aktiivisesti virheitä yksittäisistä komponenteista, jotka saattavat aiheuttaa muutoksia testattavaan järjestelmään. [Kasurinen 2013: s. 85–87.]

Hyväksymistestaus (acceptance testing) on vaihe, jonka aikana virallisesti varmennetaan tuotteen täyttävän vaatimusmäärittelyssä asetetut tavoitteet ja todetaan sen olevan tarpeeksi korkealaatuinen. Onnistuneen hyväksymistestauksen tarkoituksena on luovuttaa ohjelmisto asiakkaan omaisuudeksi, minkä seurauksena toimittajan lakiteknisen huolto- ja korjausvelvoite päättyy. Hyväksymistestaus eroaa järjestelmätestauksesta siltä osin, että hyväksymistestausta tehdään oikeassa ympäristössä eikä testausympäristössä. [Kasurinen 2013: s. 87–88.]

Savutestaus (smoke testing) on yksinkertainen tapa varmentaa, että järjestelmä käynnistyy eikä järjestelmän avainominaisuuksissa ilmene ongelmia. Savutestaus toimii periaatteessa muurina, minkä tarkoituksena on estää testiympäristön turha käynnistyminen ja välttää raskaampien testien suoritus. Koko järjestelmän kattavaa testausta ei kannata aloittaa, jos savutestien suorittamisessa ilmenee ongelmia, koska se indikoi muiden suoritettavien testien epäonnistuvan. [Laboon 2016: s. 84.]

Regressiotestauksella (regression testing) käytännössä tarkoitetaan ohjelmiston uudelleen testaamista, kun sen toimivaksi todettuihin osiin kohdistuu mikä tahansa muutos. Tarkoituksena on varmentaa, että ohjelmisto edelleen toimii odotetulla tavalla eikä muutos ole aiheuttanut haitallista vahinkoa. Regressiotestauksen tärkein ominaisuus on varmentaa, etteivät aikaisemmin ilmenneet virheet ilmaannu uudelleen eikä muutos ole vaikuttanut haitallisesti ohjelmiston uuteen versioon. [Kasurinen 2013: s. 102–103.]

Regressiotestaus on mahdollista automatisoida monilta osin, mikä tarkoittaa, että olemassa olevat automatisoidut testit suoritetaan ohjelmistoon kohdistuneen muutoksen jälkeen ja saadaan automaattisesti palautetta ohjelmiston tilasta. Automatisoidun regressiotestauksen etuihin kuuluu kehitystiimin kasvanut luottamus tiimiin ja tuotteeseen, ja se nopeuttaa kehitysvauhtia eikä testaajasta muodostu kehitystiimin ainut turvaverkko. [Crispin 2009: s. 261–262.]

2.5 Käyttöliittymäsuunnittelu ja prototyypit

Testaamista on mahdollista tehdä koko ohjelmistokehityksen ajan eikä pelkästään projektin toteutus- tai loppuvaiheessa. Varhainen testaus voidaan aloittaa jo konseptiversioilla (proof-of-concept), prototyypeillä, sovelluksen vaatimusten keräämisellä tai markkinatutkimuksella. Tätä vaihetta kutsutaan esitestaukseksi, ja sen aikana ei keskitytä testitapauksien luontiin tai toteuttamiseen, vaan sen sijaan osallistutaan ohjelmiston suunnitteluun ja konseptin validointiin. Esitestaamisen tarkoituksena on säästää rahaa poistamalla etukäteen virheitä, jotka johtuvat tuotteen puutteellisesta tuntemisesta tai ristiriitaisista vaatimuksista. [Kasurinen 2013: s. 97.]

Esitestauksen aikana ohjelmistolle suunniteltua käyttöliittymää voidaan käyttää hyödyksi, koska sen suunnittelu vaatii usein paljon huomiota ja käyttöliittymä usein sisältää kaikki mahdolliset tavat käyttää ohjelmiston ominaisuuksia. Tämän vuoksi prototyyppien laatiminen käyttöliittymän pohjalta on yksi vaihtoehto selvittää, täyttääkö suunniteltu käyttöliittymä kaikki mahdolliset tavat käyttää sovellusta.

Prototyyppien muihin etuihin kuuluu niistä saatava palaute käyttäjätestauksessa ja konseptin validoinnissa. Käyttäjät usein antavat arvokkaampaa palautetta, kun heillä on pääsy konkreettiseen asiaan, verrattuna siihen, että heiltä kysytään kuinka jonkin pitäisi toimia heidän mielestään. Prototyyppejä voidaan myös hyödyntää todellisen ohjelmiston

testitapauksien suunnittelussa ja validoinnin tukena. Niiden avulla saa vastauksia konseptin sisältämiin olettamiin, mikä vuorostaan vahvistaa tai heikentää konseptin oikeellisuutta.

Prototyyppien elinikä ei yleensä ole kovinkaan pitkä, koska niiden tarkoituksena on selvittää, eteneekö suunnittelu oikeaan suuntaan eikä niillä välttämättä sen jälkeen ole enää käyttöä. [Kasurinen 2013: s. 97–98.]

2.6 Sovelluksen rajoitettu jakelu

Sovelluksen julkaisua edeltävässä vaiheessa voidaan käyttää rajoitettua jakelua, kuten alfa- ja beeta-jakelua, joiden aikana sovelluksen kohderyhmän käyttäjät osallistetaan mukaan sovelluksen uuden version varhaiseen testaukseen. Alfa- ja beeta-jakelun aikana käyttäjiltä pyritään saamaan palautetta sovelluksesta ja sen uusista toiminnoista, ja kehitystiimillä on mahdollisuus selvittää tarkemmin loppukäyttäjän odotukset ja pyrkiä ymmärtämään, täyttävätkö sovellus ja sen toiminnot loppukäyttäjien tarpeet.

Alfa-testaus edeltää beeta-testausta, ja sen käyttö edellyttää, käyttäjien tiedostamista, että sovellus on epävakaassa tilassa ja sen aikana kerätään palautetta uusista toiminnollisuuksista eikä sovelluksessa esiintyvistä virheistä. Beeta-testauksessa puolestaan sovelluksen on tarkoitus olla stabiili ja testauksen aikana käyttäjiä pyydetään testaamaan sovellusta ja antamaan palautetta siinä esiintyvistä virheistä. [Crispin 2009: s. 465–467.]

Osa Alfa- ja beeta-jakeluun tarkoitetuista työkaluista sisältää hyödyllisiä työkaluja, joita voidaan käyttää sovelluksen kehityksen tukena. Näitä ovat esimerkiksi sovelluksen etäasennus käyttäjän laitteeseen lähes automaattisesti, kaatumis- ja virheraportointi ja mahdollisesti sovelluksen sisäinen palautteenantokanava. Osa olemassa olevista työkaluista kerää analytiikkaa sovelluksen käytöstä, ja sen avulla saadaan arvokasta lisätietoa käyttäjien tottumuksista. Sovelluksen testaajille kuitenkin tulisi tiedottaa työkaluista ja niiden keräämästä tiedosta. [Knott 2015: s. 138–139.]

Apple tarjoaa ilmaisen TestFlight-palvelun beeta-jakeluita varten. Sitä voidaan käyttää sovelluksen testaukseen rajoitetulla käyttäjämäärällä, ennen kuin sovellus julkaistaan App Store -kauppaan kaikkien saataville. Jakelut on mahdollista kohdistaa erillisille kohderyhmille, jotka voidaan jakaa karkeasti sisäisiin ja ulkoisiin testaajiin.

Sisäiset testaajat ovat osa kehittäjän tiimiä iTunes Connect-hallinnointiportaalissa, jossa heille on asetettu tietty rooli, kuten esimerkiksi tilin hallinnoija, sovelluskehittäjä tai markkinoija. Sisäisiä testaajia voi kuitenkin olla vain 25 jakelua kohden, mikä on vähemmän kuin mahdollisten ulkoisten testaajien sallittu enimmäismäärä. Ulkoisia testaajia voi olla enintään 10 000. He ovat nimensä mukaisesti kehitystiimin ulkopuolisia henkilöitä. [TestFlight beta testing overview (iOS, tvOS, watchOS); Testing Apps with TestFlight]

Käytännössä beeta-testaukseen kutsuttu henkilö saa sähköpostiinsa kutsun, jossa henkilöä kehotetaan asentamaan TestFlight-sovellus, jonka avulla beeta-testattava sovellus asennetaan käyttäjän laitteeseen. [Beta Testing Made Simple]

Crashlytics Beta on toinen ilmainen vaihtoehto beeta-jakeluita varten. Sen avulla voidaan tehdä beeta-jakeluita Android- ja iOS-alustalle. Crashlytics Beta -työkalut on mahdollista lisätä osaksi kehittäjän editoria ladattavissa olevilla laajennuksilla, joiden avulla sovellusjakeluiden toimitus testaajille on helppoa ja suoraviivaista. [St. Pierre: 2014]

2.7 Pilvitestaustyökalut

Pilvitestaustyökalut tarjoavat monipuolisen pilvessä sijaitsevan testilaitekokoelman, joka koostuu erilaisista laitemalleista ja käyttöjärjestelmäversioista, jotka voivat olla oikeita fyysisiä laitteita, emulaattoreita tai simulaattoreita. Pilvessä sijaitsevilla laitteilla voi tehdä funktionaalisia, suorituskyky-, kuormitus- tai laitekohtaista testausta. Käytännössä sovelluksen testaaminen pilvessä sijaitsevalla laitteella tai useammalla laitteella noudattaa tiettyä kaavaa. Sovellus ladataan pilvipalveluun, minkä jälkeen palvelu asentaa sovelluksen valitulle laitteelle ja käynnistää sen laitteessa. Sovelluksen käynnistymisen jälkeen haluttu testitapaus suoritetaan manuaalisesti tai automaattisesti. Pilvessä sijaitseva palvelu voi tarjota mahdollisuuden nauhoittaa tai ottaa kuvakaappauksia testitapauksen suorituksesta, mitkä voivat osoittautua hyödylliseksi ongelmien selvittelyssä.

Pilvitestaustyökalujen vahvuuksiin kuuluu pääsy palveluun mistä tahansa, monipuolinen laitesaatavuus ja testien suorittaminen rinnakkain useilla eri laitteilla sekä se, että kehitystiimin tarvitse ostaa useita fyysisiä laitteita testausta varten. Heikkoutena voidaan pitää pilvessä testien hidasta suoritusnopeutta, laitteen sensoreiden ja muiden ominaisuuksien rajoitettua käyttöä sekä palvelun käytöstä aiheutuvia kuluja. Tietoturvan näkökulmasta huolestuttavin seikka on, poistetaanko palveluun ladattu sovellus testilaitteesta

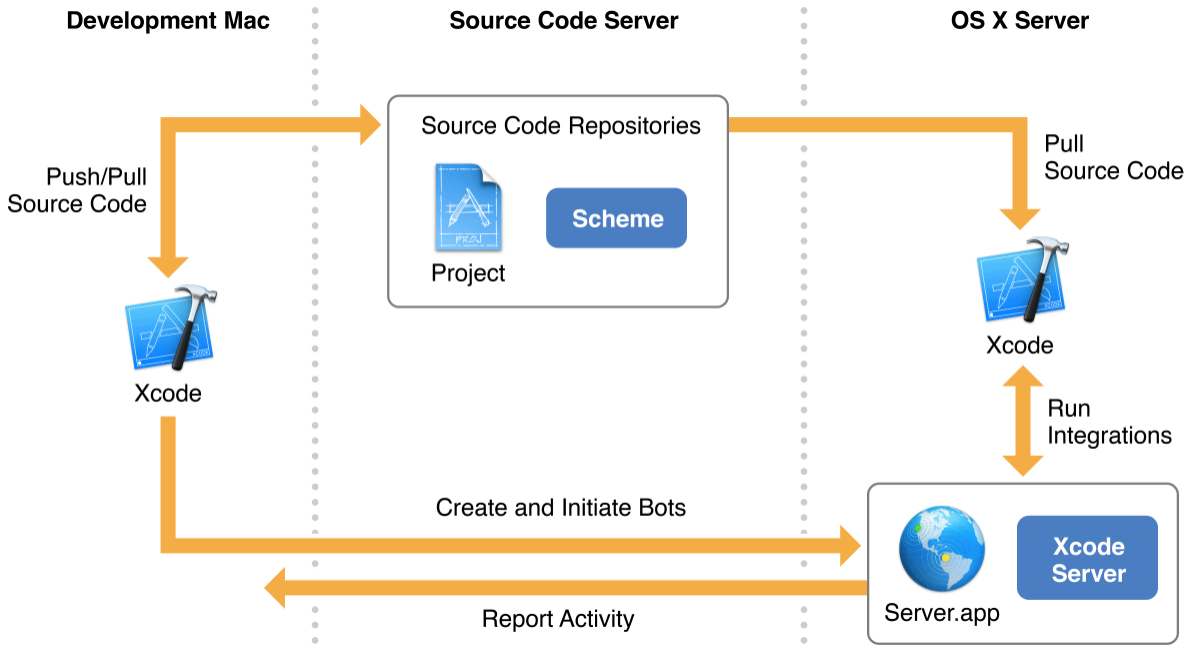
tai ylipäätään pilvipalvelusta, minkä laiminlyöminen saattaa mahdollistaa pääsyn sovellukseen palvelun muille käyttäjille.

Pilvitestaustyökalut voivat olla julkisen palveluntarjoajan tarjoamia, jolloin ne ovat vapaasti kaikkien palvelua käyttävien käytössä. Vaihtoehtoisesti pilvipalvelu voidaan hankkia yksityiseen käyttöön ja se voi sijaita kehitystiimin omissa tiloissa, jolloin pääsy pilvipalveluun ja sen laitteistoon on rajattu. Valinta julkisen tai yksityisen pilvipalvelun käyttöönotolle riippuu yrityksen tai sovelluksen tietoturvavaatimuksista. [Knott 2015: s. 147–151.]

2.8 Jatkuva integraatio

Jatkuva integraatio (CI) on ohjelmistokehityksen kannalta tärkeä työkalu, jonka avulla saadaan nopeaa palautetta kehitystiimille ohjelmiston senhetkisestä tilasta, ja se helpottaa ohjelmiston ylläpitämistä julkaisukelpoisessa kunnossa. Se on käytännössä palvelimella sijaitseva järjestelmä, joka koostuu erilaisista suoritettavista automatisoiduista vaiheista, joita voivat olla muun muassa lähdekoodin analysointi, sovelluksen koontiversion teko, automatisoitujen testien suorittaminen ja sovelluksen beeta- tai julkaisuversion toimitus jakelujärjestelmään. Tyypillisesti CI-järjestelmä suorittaa automatisoidut vaiheet versionhallintaan kohdistuneen muutoksen yhteydessä, mutta sen voi ajastaa käynnistymään tiettyinä ajanhetkinä tai sen voi käynnistää manuaalisesti. [Knott 2015: s. 138–139; About Continuous Integration in Xcode]

Applen dokumentaation mukaan jatkuvan integroinnin työnkulku (kuva 4) Xcode-editorissa koostuu seuraavista komponenteista: kehittäjän paikallinen kehitysympäristö, versionhallintajärjestelmä ja OS X -palvelin. Paikallisessa kehitysympäristössä kehittäjä voi luoda, muokata tai poistaa bot-komponentteja, jotka sijaitsevat OS X -palvelimella. Bot-komponentit suorittavat niille määritettyjä työvaiheita tietyissä tilanteissa, ja prosessia kutsutaan integraatioksi. [About Continuous Integration in Xcode]



Kuva 4. Xcoden CI-prosessin työkulku visualisoituna [About Continuous Integration in Xcode]

Käytännössä työnkulku menee seuraavalla tavalla: kehittäjä tekee muutoksen paikallisessa työympäristössä ja muutos viedään versionhallintajärjestelmään, johon OS X-palvelimella sijaitsevat Bot-komponentit reagoivat. Mikäli niille määritetyt ehdot täyttyvät, Bot-komponentti hakee ensimmäiseksi versionhallinnasta uusimman version sovelluksesta, suorittaa integraation ja sen päättyessä antaa palautetta kehittäjän editoriin tai muuhun määritettyyn paikkaan. [About Continuous Integration in Xcode]

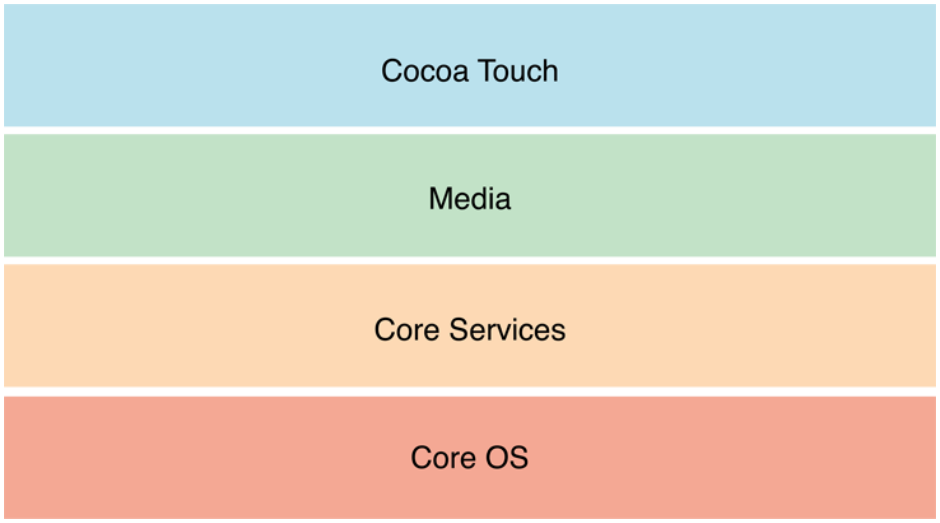
CircleCI-pilvipalvelu on vaihtoehto edellä mainitulle CI-järjestelmälle, se tarjoaa pilvipalveluna CI-ympäristön, jossa voidaan suorittaa automatisoituja testejä ja automatisoitu sovelluksen toimitus haluttuun paikkaan onnistuneen integroinnin jälkeen. [Lazinskiy 2017]

3 iOS-käyttöjärjestelmä ja -kehitysympäristö

3.1 iOS-käyttöjärjestelmä

iOS on Applen kehittämä käyttöjärjestelmä, jota käytetään muun muassa iPhone- ja iPad-laitteilla. Käyttöjärjestelmän tarkoituksena on ylläpitää laitteen toimintoja ja tarjota ympäristö, jossa voidaan suorittaa natiivisovelluksia. Käyttöjärjestelmän mukana tulee käyttäjälle yleishyödyllisiä natiivisovelluksia, kuten erillinen sähköposti- tai verkkoselain-sovellus. Natiivisovelluksia voi rakentaa käyttämällä iOS SDK:ta (Software Development Kit), joka sisältää kehittäjän tarvitsemat työkalut ja rajapinnat, joita tarvitaan sovelluksien kehittämiseen, testaamiseen ja suorittamiseen.

iOS-käyttöjärjestelmän arkkitehtuuri on jaettu eri tasoihin. Arkkitehtuurin eri tasot tarjoavat sovelluksen käyttöön erilliset hyvin määritetyt rajapinnat, joita sovellus voi käyttää sen ja laitteen väliseen kommunikaatioon. Kuvassa 5 esitetyt tasot toimivat viestinvälittäjinä laitteen ja sovelluksien välillä.



Kuva 5. iOS-arkkitehtuurin eri tasot [Apple Developer Documentation].

Hyvin määritetyt rajapinnat helpottavat sovelluksien kehittämistä teknisesti erilaisille laitteille. Alemmat tasot pitävät sisällään käyttöjärjestelmän tärkeimpiä toimintoja ja palveluita, joita ylemmän tason toteutukset hyödyntävät. Ylemmän tason alueet tarjoavat kehittyneempiä palveluita ja teknologioita kehittäjän käyttöön. [Apple Developer Documentation]

3.2 Kehitysympäristö

Sovelluskehitys Applen alustoille vaatii Mac-tietokoneen, päivitetyn version OS X -käyttöjärjestelmästä ja Xcode-ohjelmointiympäristön [Signing workflow].

Apple on kehittänyt omaa käyttöjärjestelmää jo kymmenien vuosien ajan. Ensimmäisen käyttöjärjestelmän nimi on System, ja sitä seurasi vuosien jälkeen OS X, joka nykypäivänä tunnetaan nimellä macOS. macOS on Unix-pohjainen käyttöjärjestelmä, joka toimii Applen pöytätietokoneissa ja kannettavissa malleissa. [Nelson 2017]

Xcode ohjelmointiympäristö on keskeinen osa Applen laitteille tehtävässä sovelluskehityksessä. Ohjelmointiympäristöön on integroitu tarvittavat sovelluskehykset ja työkalut, joita tarvitaan Mac-, iPhone-, iPad-, Apple Watch- ja Apple TV -laitteille suunnatussa sovelluskehityksessä. [Tools you'll love to use]

Swift on moderni, avoimeen lähdekoodiin perustuva ohjelmointikieli, jota on mahdollista käyttää monipuolisesti eri ympäristöissä, kuten mobiili-, työpöytä- ja pilvisovellusten kehityksessä. Kirjoitushetkellä Swiftiä käytetään mm. macOS-, iOS-, watchOS- ja tvOS-sovelluskehityksessä ja sitä voi suorittaa Linux-ympäristössä. Ohjelmointikielen kehityksessä kielen turvallisuus, nopeus ja ilmaisuvoimaisuus ovat tärkeitä pilareita, ja nämä ominaisuudet on pyritty rakentamaan sisään ohjelmointikieleen sen kehityksen aikana.

Swiftin turvallisuus käy ilmi kielen asettamien rajoitteiden muodossa: niillä pyritään välttämään sovelluksen epämääräistä käyttäytymistä ennalta arvaamattomien tilojen seurauksena. Kielen rajoitteiden tarkoituksena on muun muassa estää alustamattomien muuttujien käyttö, kokonaislukujen ylivuoto ja muistinkäytön hallinnointi kehittäjän puolesta. Swiftin tarkoitus on toimia useiden C-kielien manttelinperijänä, minkä vuoksi sen suoritusnopeuden täytyy olla hyvä, koska sitä tarvitaan vaativissa laskentatehtävissä. Swiftin ilmaisuvoimaisuus näkyy miellyttävänä syntaksina ja moderneina toimintoina sekä yhteisön haluna kehittää kieltä paremmaksi. [About Swift]

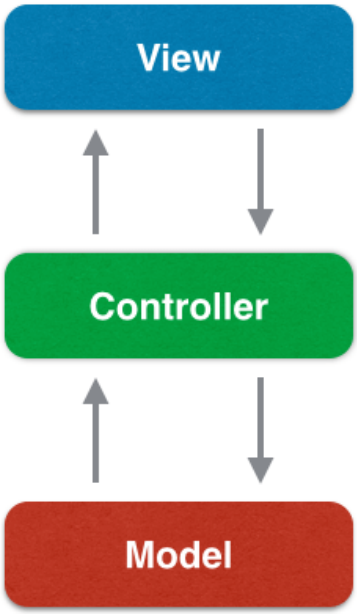
3.3 iOS-sovelluksen arkkitehtuuri

Ohjelmistoarkkitehtuurin tehtävänä on määrittää ratkaisusta kuvaus, joka täyttää sovellukselle asetetut toiminnalliset ja tekniset vaatimukset. Ohjelmisto tulisi rakentaa luotettavalle pohjalle, ja ohjelmiston tärkeimmät piirteet tulisi huomioida suunnitteluvaiheessa, jotta vältytään tulevaisuudessa tarpeettomilta ongelmilta. Huonosti soveltuvan ohjelmistoarkkitehtuurin valinnalla voi olla ikäviä seurauksia, kuten ohjelmiston epävakaus, vaikea ylläpidettävyys tai se, ettei se täytä tulevaisuuden vaatimuksia. [What is Software Architecture?]

On olemassa useita arkkitehtuuriin vaikuttavia malleja ja tyylejä, joita kutsutaan yleisemmin arkkitehtuurityyleiksi (architectural styles). Ne sisältävät korkeatasoisen kuvauksen tavasta ratkaista yleisiä sovellusarkkitehtuuriin liittyviä ongelmia. Eri arkkitehtuurityylit eivät ole toisiaan poissulkevia, vaan niitä on mahdollista käyttää samanaikaisesti.

Kerroksittainen arkkitehtuuri (layered architecture) on yksi näistä arkkitehtuurityyleistä, ja sen tarkoituksena on jakaa ohjelmisto eri tasoihin vastuualueiden mukaan, mikä vuorostaan parantaa ohjelmiston joustavuutta ja ylläpidettävyyttä. Tasojen välinen kommunikointi tapahtuu eksplisiittisesti, ja niiden rajapinnat ovat löyhästi sidottu toisiinsa. Useat eri suunnittelumallit käyttävät kyseistä arkkitehtuurityyliä, ja yksi näistä on Separated Presentation -malli, jossa sovelluksen logiikka jaetaan kolmeen toisista eroavaan tasoon tai rooliin vastuualueiden mukaan. Yksi mainittua menetelmää noudattavista suunnittelumalleista on nimeltään Model-View-Controller (MVC). [Architectural Patterns and Styles]

MVC (Model-View-Controller), on yleisesti käytetty suunnittelumalli, joka jakaa ohjelmiston vastuualueet kolmeen eri tasoon (kuva 6), jotka ovat malli (model), näkymä (view) ja kontrolleri (controller). Useat Cocoa-pohjaiset teknologiat ja arkkitehtuurit käyttävät MVC-suunnittelumallia, ja niiden käyttöönotto edellyttää kehittäjältä jonkin MVC-tason varsinaista toteutusta. [Model-View-Controller]



Kuva 6. Model-View-Controller ja sen eri tasojen välinen kommunikointi [Manferdini 2016].

Model-tason objektit sisältävät sovelluksessa esitettävän sisällön ja toimintalogiikan, joka määrittää kuinka sovelluksessa esitettävä sisältö käsitellään ja esitetään. Tasossa sovelluksen data voidaan tallentaa laitteen tiedostojärjestelmään tai sovelluksen sisäiseen tietokantaan. Model-tason objektit ovat usein uudelleenkäytettävissä, koska ne kuvaavat sovelluksessa esitettävää tietoa ja sisältävät toimintalogiikan tiedon esittämiseen.

View-tason vastuualueisiin kuuluu sovelluksen sisällön esittäminen ja se antaa käyttäjän muuttaa sovelluksen sisältöä käyttöliittymästä. View-taso käytännössä esittää Model-tason objektin tai objektien sisältämän tiedon sovelluksen käyttöliittymässä.

Controller-taso toimii kahden muun kerroksen välisenä välikätenä, jonka kautta muut tasot kommunikoivat. Sen tehtäviin kuuluvat sovelluksen erilaiset alustus- ja tiedonvälitystehtävät, joita esimerkiksi View-taso käyttää saadakseen Model-tasosta esitettävän sisällön. [Model-View-Controller]

3.4 Tunnetuimmat alustakohtaiset testikirjastot

XCTest on Xcode-ohjelmointiympäristöön integroitu testikehys. Sen avulla voidaan kirjoittaa yksikkö-, suorituskyky- tai käyttöliittymätestejä, joita voidaan suorittaa simulaattorissa tai oikeassa laitteessa. [XCTest Documentation; Helppi 2016b]

XCTest-kehyksellä kirjoitetut testit noudattavat ennalta määritettyä rakennetta, jossa yhden luokan sisälle kirjoitetaan haluttu määrä testitapauksia testattavaa luokkaa kohden. Luokan sisältämien testitapauksien metodit noudattavat tiettyä rakennetta ja nimeämistapaa, jossa metodin nimen tulee sisältää prefiksinä test, eikä metodi palauta mitään. Poikkeuksena ovat erilliset testausta tukevat metodit, joilla testi voidaan alustaa tai sen suorituksen jälkeinen tila voidaan siivota.

Testattavan luokan, funktion tai metodin lopputulos voidaan verifioida kehyksen tarjoamilla XCTAssert-tarkisteilla. Tarkisteilla vertaillaan odotetun ja saadun lopputuloksen objektien yhdenmukaisuutta tai erilaisia loogisia lausekkeita, joilla varmistutaan halutusta lopputuloksesta.

XCTest-kehyksellä voi laatia komponenttien suorituskykytestejä, joiden avulla mitataan testattavan komponentin suoritukseen kuluva aika. Suorituskykytestauksen avulla ylläpidetään testauksen kohteen tavoiteltua performanssia tarkastelemalla testattavan komponentin senhetkistä suoritusnopeutta, jota verrataan aikaisempien testien suoritusnopeuksiin. Suorituskykytestauksessa regressio ilmenee, kun testattavan kohteen suoritusnopeus on ajallisesti pidempi kuin aikaisemmin saatu pidemmän aikavälin keskiarvo. [Testing Basics]

Sovelluksen käyttöliittymätestaus pohjautuu kahteen keskeiseen teknologiaan, XCTest-kehykseen sekä iOS- ja macOS- alustan helppokäyttöisyystoimintoihin. XCTest tarjoaa käyttöliittymätestaukselle pohjan, jonka avulla voidaan kirjoittaa testejä ja varmentaa testitapauksien haluttu lopputulema. Helppokäyttöisyystoiminnot ja niiden rajapinnat mahdollistavat käyttöliittymän testauksen, koska helppokäyttöisyystoimintojen rajapinnat tarjoavat testejä varten tarvittavat tiedot käyttöliittymän komponenteista.

Käyttöliittymätestaus eroaa yksikkö- ja suorituskykytestauksesta testaamisen tason ja testaamistavan osalta. Yksikkö- ja suorituskykytesteillä testataan enimmäkseen sovelluksen sisäisiä osia, kun taas käyttöliittymätestien tarkoituksena on simuloida käyttäjän

suorittamia tapahtumia ja varmentaa käyttöliittymän tila halutuksi. Käytännössä käyttöliittymätestauksen aikana paikannetaan käyttöliittymässä esiintyviä samoja ongelmia, joita loppukäyttäjät kokisivat. [Testing Basics]

Käyttöliittymätestaus noudattaa yksinkertaisimmillaan seuraavaa rutiinia:

- Paikanna käyttöliittymän elementti.
- Verifioi elementin senhetkinen tila tai käyttäytyminen.
- Suorita käyttöliittymässä jokin tapahtuma, joka muuttaa sen tilaa.
- Verifioi käyttöliittymän tila halutuksi. [Testing Basics]

Quick on käyttäytymispohjainen testikirjasto Swift- ja Objective-C-ohjelmointikielille. Käyttäytymispohjaisten testien etuihin kuuluu, että ei-tekniset henkilöt kykenevät tulkitsemaan laadittuja testejä ja testejä voidaan käyttää laadunvarmennuksessa. Vaatimusmäärittelyä voidaan käyttää apuna käyttäytymispohjaisen testin laatimisessa, koska konkreettiset testitapaukset voidaan kopioida määrittelystä ja saadaan varmuus vaatimusten täyttymisestä. Quick sisältää oletuksena Nimble-kirjaston, joka täydentää Quick-testeissä käytettävää syntaksia. [Zablocki 2017; Quick and Nimble Testing with Swift]

Quick sisältää kolme erillistä avainsanaa, jotka ovat describe, context ja it. Jokainen avainsana luo erillisen sulkeuman ja niillä on testissä dedikoitu käyttötarkoitus. Describe-avainsana kuvailee testauksen kohteen tai mitä pitäisi tapahtua. Context-avainsana kertoo, minkälainen testattavan kohteen tilan, sen pitäisi olla ennen varsinaista väitteen verifiointia. It-avainsana kertoo, mitä testissä tapahtuu, ja avainsana toimii tunnisteena, jota käytetään testin epäonnistuessa. [Quick and Nimble Testing with Swift]

Nimble-testikirjastolla voidaan kirjoittaa monipuolisia saadun ja halutun tuloksen vertailijoita (matcher) tai käyttää kirjastossa jo olemassa olevia funktioita tai makroja vertailuun. Nimble-kirjaston valmiit funktiot ja syntaksi helpottavat koodin lukemista ja ymmärrystä verrattuna perinteiseen XCT-kehyksellä tehtyihin väitteisiin. Toisena Nimble-kirjaston etuna voi pitää, että se helpottaa asynkronisten testien laatimista verrattuna perinteiseen XCTest-kehykseen. [Quick; Quick and Nimble Testing with Swift]

EarlGrey on Googlen sisäisesti kehittämä ja käyttämä testauskirjasto, jota Google on käyttänyt useiden iOS-sovellusten testauksessa. Se on tarkoitettu käytettäväksi käyttöliittymän funktionaalisessa testauksessa, ja se on hyvin samankaltainen kuin Googlen kehittämä Espresso-testikirjasto Android-sovellusten testausta varten. EarlGrey-kirjaston vahvuus piilee sen sisäänrakennetussa synkronoinnissa, käyttöliittymän komponenttien monipuolisissa tarkastuksissa ja kirjaston joustavassa suunnittelussa. [Helppi 2016a; Janga 2016]

Sisäänrakennettu synkronointi vaikuttaa monipuolisesti iOS-sovellusten testaamiseen, se helpottaa käyttöliittymän, verkkopyyntöjen ja sovelluksen muiden jonojen käsittelyssä. Sisäänrakennettu synkronointi takaa sen, että käyttöliittymä on vakaassa tilassa ennen kuin testiskriptin sisältämät toiminnot suoritetaan, mikä parantaa testien vakautta ja toistettavuutta. Kolmantena etuna voidaan pitää sitä, että testaaja näkee laitteen tai simulaattorin käyttöliittymän avulla, mitä suoritettavassa testitapauksessa tapahtuu vaihe vaiheelta. [Helppi 2016a; EarlGrey]

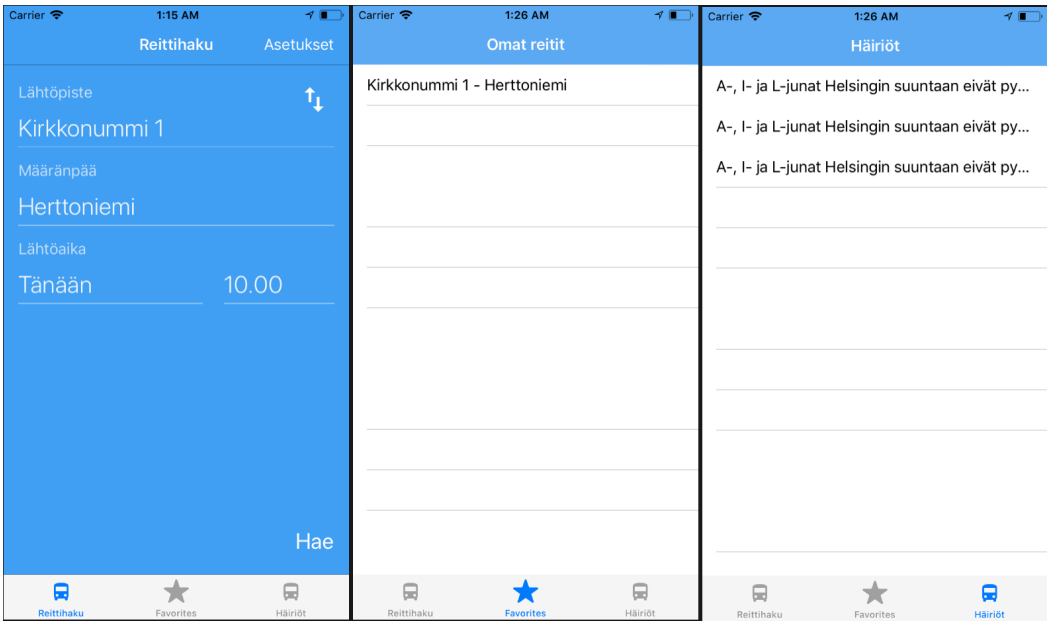
Käyttöliittymälle tehtävien interaktioiden rakenne noudattaa tiettyä kaavaa, johon kuuluvat element_matcher, action ja assertion_matcher. Ensimmäisen, eli element_matcherin, tarkoituksena on paikantaa jokin tietty käyttöliittymän elementti tai komponentti, kuten nappi tai teksti. Toisena listatun actionin tarkoituksena on suorittaa jokin toimenpide ensimmäiselle käyttöliittymän kohteelle, kuten esimerkiksi painallus tai tekstinsyöttö. Viimeisenä on assertion_matcher, jolla varmistetaan, että elementin tila on pysynyt ennallaan tai muuttunut jollakin tavalla, kuten näkymättömästä näkyväksi. [EarlGrey]

4 Sovelluksen kehitys ja testaus

4.1 Yleistä

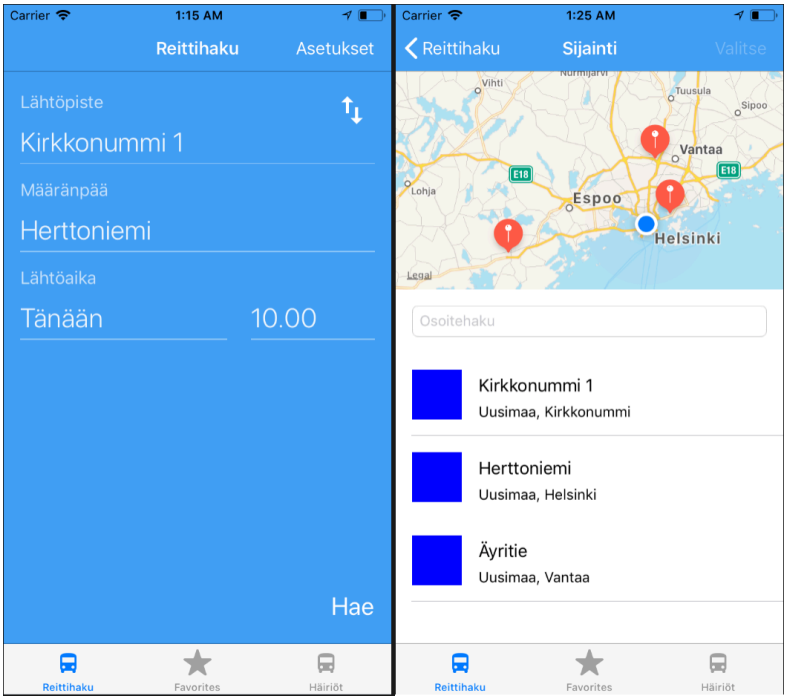
Insinöörityössä tehdyn iOS-sovelluksen avulla voi hakea Helsingin kaupungin Reittioppaan rajapinnasta kaupungin sisäisen liikenteen reittejä haluamanaan ajankohtana. Sovelluksen avulla voi hakea reittitiedot kahden sijainnin välillä käyttäjän määrittämään ajankohtaan, tehdä reittihaun omista reiteistä ja näyttää liikennöintiä koskevat häiriötilanteet.

Sovelluksen ylimmän tason navigaatio (kuva 7) koostuu reittihausta, suosikeista ja häiriönäkymästä.



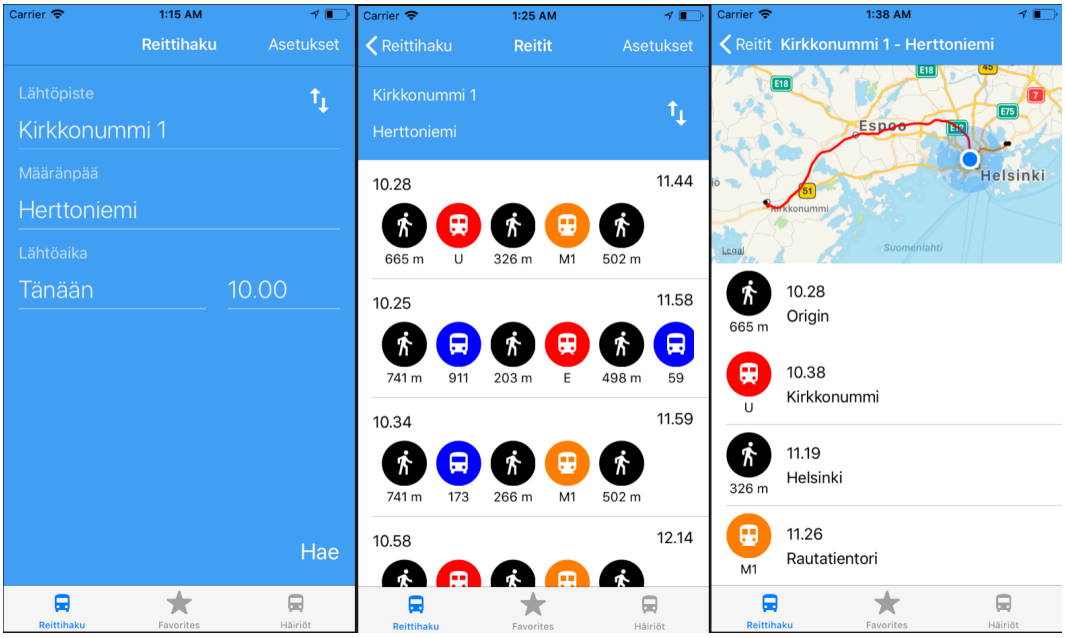
Kuva 7. Sovelluksen ylimmän tason navigaatio, reittihaku, suosikit ja häiriöt.

Reittihaku-näkymästä voi siirtyä Oma sijainti -näkymään, josta voi hakea käyttäjän senhetkisen sijainnin tai käyttäjä voi suorittaa osoitehaun sijaintinäkymässä (kuva 8) olevaan tekstikenttään, joka hakee hakutulokset reittioppaan tarjoamasta geokoodaus-rajapinnasta.



Kuva 8. Reittihaku ja sijaintinäkymät.

Sovelluksen reittihaku hakee viisi seuraavaa reittiä lähtöpisteestä määränpäähän. Reitit-näkymän yksittäistä lista-alkiota painamalla käyttäjä siirtyy reitin detaljinäkymään (kuva 9), josta lista-alkiota painamalla karttakomponentti zoomaa kyseiseen reitin matkavaiheeseen.

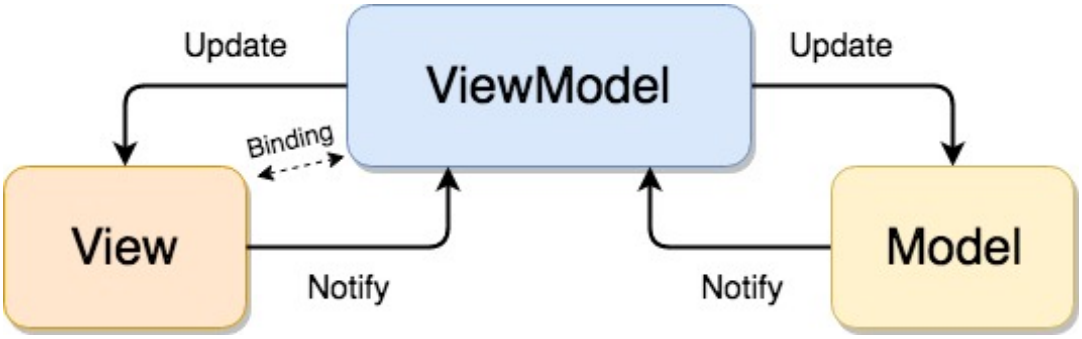


Kuva 9. Reitin haku, tulokset ja detaljinäkymä, jossa vaiheet näkyvät erikseen kartalla.

4.2 Sovelluksen rakenne ja toteutus

Reittihaku-sovellus kehitettiin Xcode-kehitysympäristössä, ja sovelluksen minimiversioksi on asetettu kirjoitushetkellä tuorein iOS-versio, joka on 11.2. Sovelluksen käyttöliittymä rakennettiin lähestulkoon kokonaan Storyboard-tiedostomallilla, jossa määritellään sovelluksen navigointilogiikka, yksittäiset näkymät ja niiden käyttöliittymäkomponentit. Yksinkertaistamisen ja ajan säästön vuoksi sovellus ei tue vaakataso-orientaatiota eikä tableteille erillistä käyttöliittymää.

Sovelluksen arkkitehtuurimallina käytettiin Model-View-ViewModel (MVVM) -arkkitehtuurimallia (kuva 10). Valintaan vaikutti reaktiivisten kirjastojen käyttö sovelluksessa, niiden käyttö soveltuu hyvin käytettyyn arkkitehtuurimalliin, jossa ViewModel ja View ovat kytketty toisiinsa käyttämällä reaktiivista kirjastoa. Sen käyttö mahdollistaa automaattisen tavan päivittää ja ilmoittaa muutoksista komponenttien välillä.



Kuva 10. Model-View-ViewModel-arkkitehtuurimalli [Santarossa]

Käytännössä View ilmoittaa käyttöliittymässä tapahtuneesta muutoksesta ViewModel-luokalle, joka päivittää tarvittaessa Model-luokan. Sen jälkeen Model-luokka ilmoittaa muutoksesta ViewModel-luokalle, joka puolestaan päivittää käyttäjälle esitettävän näkymän. View- ja ViewModel-luokkien väliseen kytkemiseen käytettiin reaktiivista kirjastoa, jonka avulla eri komponentit reagoivat nimensä mukaisesti uusiin tapahtumiin tai muutoksiin.

Sovelluksen käyttöliittymän koostamisessa käytettiin pääasiassa UIKit- ja MapKit-kehikkoa ja niiden tarjoamia käyttöliittymäkomponentteja. Kartassa näkyvän reitin piirtämisessä käytettiin Polyline-kirjastoa, jonka avulla voi dekoodata rajapinnan palauttaman reittidatan ja antaa sen syötteenä karttakomponentille, joka piirtää reitin käyttäjälle ymmärrettävään muotoon. Kolmannen osapuolen SVProgressHud-kirjastoa käytettiin latausindikaattorin näyttämiseen käyttöliittymässä.

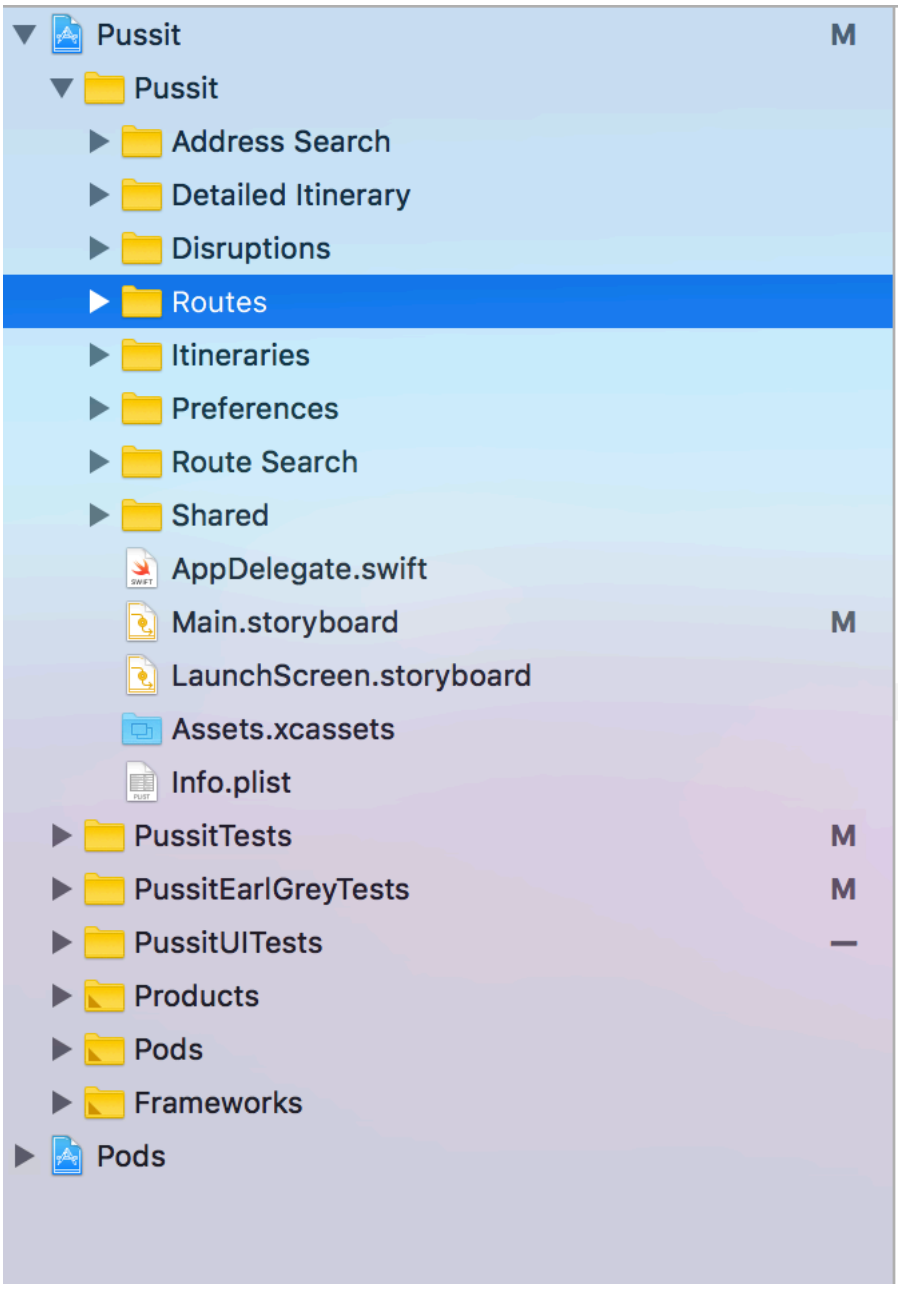
Sovelluksen kehityksessä käytettiin laajasti reaktiivisia kirjastoja: RxSwift-kirjastoa ja sen Cocoa-ympäristölle tehtyä laajennosta RxCocoa-kirjastoa. RxCocoa sisältää valmiita toteutuksia UIKit-kehikon sisältämille komponenteille, kuten tekstikentälle tai listalle. Sovelluksen verkkopyyntöjä varten käytettiin Alamofire-kirjastoa, jonka avulla oli vaivatonta tehdä verkkokutsuja reittioppaan avoimeen rajapintaan. Rajapinnan JSON-vastauksien käsittelyyn käytettiin SwiftyJSON-kirjastoa.

Sovelluksen kirjastoriippuvuuksien hallintaan käytettiin CocoaPods-työkalua. Kirjastoriippuvuudet lisätään Podfile-tiedostoon, joka sijaitsee sovelluksen kansiorakenteen juuressa. Podfile sisältää sovelluksessa käytettävät riippuvuudet (esimerkkikoodi 1), jotka ovat edellä mainitut Alamofire, RxCocoa, RxSwift, SwiftyJSON, Polyline ja SVProgresshud.

```
platform :ios, '8.0'
use_frameworks!
target 'Pussit' do
     pod 'Alamofire',     '~> 4.5'
     pod 'RxCocoa',       '~> 4.0'
     pod 'RxSwift',       '~> 4.0'
     pod 'SwiftyJSON',    '~> 4.0'
     pod 'Polyline',      '~> 4.0'
     pod 'SVProgressHUD','~> 2.2'
end
target 'PussitTests' do
     pod 'Alamofire',  '~> 4.5'
     pod 'RxCocoa',    '~> 4.0'
     pod 'RxSwift',    '~> 4.0'
     pod 'SwiftyJSON', '~> 4.0'
     pod 'Nimble',     '~> 7.0'
     pod 'Quick',      '~> 1.2'
end
target 'PussitUITests' do
     pod 'Nimble',     '~> 7.0'
     pod 'Quick',      '~> 1.2'
end
target 'PussitEarlGreyTests' do
     pod 'EarlGrey', '~> 1.12'
     pod 'Nimble',     '~> 7.0'
     pod 'Quick',      '~> 1.2'
end
```

Esimerkkikoodi 1. Podfile-tiedostossa määritetyt riippuvuudet.

Sovelluksen projektirakenne (kuva 11) on jaettu sovelluksen toiminnollisuuksien mukaan niin, että jokaista toimintoa varten on Pussit-kansion alla erillinen kansio, joka sisältää todelliset toteutukset. Testit ovat puolestaan jaettu PussitTests-, PussitEarlGreyTests- ja PussitUITests-kansioihin. PussitTests-kansio sisältää sovelluksen XCTest-testikehyksellä laaditut yksikkötestit, PussitUITests-kansio sisältää XCTest-kehyksellä laaditut käyttöliittymätestit ja lopulta PussitEarlGreyTest-kansion alla on nimensä mukaisesti EarlGrey-kirjastolla toteutetut testit.



Kuva 11. Pussit-sovelluksen rakenne.

4.3 Reittiopas-rajapinnan käyttö sovelluksessa

Reittiopas tarjoaa modernin yleiseen käyttöön tarkoitetun rajapinnan reittitieto-, linjasto- ja aikataulu-kyselyitä varten osoitteessa https://digitransit.fi. Kyselyt rajapintaan tehdään käyttämällä REST-rajapintaa, joka ottaa vastaan kyselyitä GraphQL-formaatissa ja palauttaa vastauksen JSON-formaatissa. Reittitietojen, linjastojen ja aikataulujen lisäksi reittiopas tarjoaa geokoodausta varten REST-rajapinnan, jonka avulla koordinaateilla tai osoitteella voi hakea lähimmät asemat ja pysäkit.

GraphQL-kyselyiden avulla (esimerkkikoodi 2) voidaan rajoittaa kyselyn palauttamaa JSON-sisältöä, mikä tekee siitä dynaamisen ja kevyen tavan siirtää tietoa sovelluksen ja palvelimen välillä.

```
plan(from: {lat: 60.2019155, lon: 24.9431893}, to: {lat: 60.1447449, lon:
25.0238625}, numItineraries: 1) {
    itineraries {
      duration
      startTime
      endTime
      walkTime
      walkDistance
    }
  }
}
```

Esimerkkikoodi 2. Yksinkertaistettu versio sovelluksessa käytettävästä GraphQL-kyselystä reittioppaan rajapintaan.

Esimerkissä määritetään Reittioppaan rajapintaan tehtävä kysely, jonka Reittioppaan rajapinta käsittelee ja palauttaa kyselyn mukana määritetyt vastauskentät JSON-formaatissa. Käytännössä rajapinnan vastaus sisältää itineraries-sulkeaman sisältämät kentät, jotka ovat duration, startTime, endTime, walkTime ja walkDistance.

4.4 Sovelluksen testaus

Sovelluksen testauksessa käytettiin manuaalista testausta, automatisoituja testejä ja jatkuvaa integraatiota. Alfa- tai beeta-testausta ei käytetty lainkaan, koska sovellus jää lähinnä omaan käyttöön eikä tarkoituksena ole jatkokehittää sovellusta täysin julkaisukelpoiseksi.

Manuaalista testausta käytettiin sovelluskehityksen aikana jatkuvasti sovelluksen ulkoasun, käytettävyyden, keskeytystilanteista palautumisen ja käyttäjän GPS-sijainnin testauksessa. Käyttöliittymän kehityksen kannalta manuaalinen testaus toimi suhteellisen nopeasti, koska sovellus on yksinkertainen eikä se vaadi sisäänkirjautumista eikä testauksen aikana tarvinnut kirjoittaa käyttöliittymään hankalia syötteitä, jotka olisivat hidastaneet sovelluksen manuaalista testaamista. Keskeytystilanteiden testaamisessa lähinnä testattiin, että sovelluksesta poistuessa ja palatessa se jatkaa siihen palatessa odotetusti. Käyttäjän GPS-sijainnin testaamisessa manuaalinen testaus säästi eniten aikaa, koska sen automatisoitu testaaminen on melko haastavaa toteuttaa simulaattorilla.

Automatisoituja yksikkötestejä kirjoitettiin sekalaisesti Nimble-, Quick- ja XCTest-testauskirjastojen avulla, vaikka tavallisesti on järkevää pysyä tietyssä tavassa luoda testitapauksia. Normista poikettiin kirjastoihin perehtymisen vuoksi.

Esimerkkikoodista 3 käy ilmi Nimble- ja XCTest-kirjastolla kirjoitettujen testien rakenne, jossa yksittäinen metodi on käytännössä yksi yksikkötesti. Luokassa on määritetty kaksi yksikkötestiä, joissa prefiksi test määrittää niiden olevan testejä ja test-sanan jälkeinen teksti kuvaa testattavaa toiminnallisuutta. Yksikkötesteistä käyvät ilmi Nimble-kirjaston tarjoamat expect- ja to-funktiot, jotka helpottavat testien luettavuutta.

```
class UtilsTest: XCTestCase {

  func testShouldReturnColorForTransportationMode() {
    let expectedColors = [UIColor.blue, UIColor.green, UIColor.red, UI-
Color.orange, UIColor.brown, UIColor.black]
    let modes = TransportationMode.values

    expect(expectedColors.count).to(equal(modes.count))
    modes.forEach { expect(expectedColors).to(contain($0.getColor())) }
  }

  func testShouldConvertCoordinateToAnnotation() {
    let coordinate = CLLocationCoordinate2D(lat: 10, lon: 11)
    let annotation = coordinate.toMKPointAnnotation()
    expect(annotation.coordinate.latitude).to(equal(coordinate.latitude))
    expect(annotation.coordinate.longitude).to(equal(coordinate.longitude))
  }
}
```

Esimerkkikoodi 3. Yksikkötesti, jossa on käytetty XCTest- ja Nimble-kirjastoja.

Esimerkkikoodissa 4 on laadittu Nimble- ja Quick-kirjastoja käyttämällä. Niillä saatiin helpommin luettavia ja ymmärrettäviä yksikkötestejä. Yksikkötesteissä käytännössä selitetään testien toiminta kuvaavilla avainsanoilla ja niiden sisältämillä kuvauksilla. Describe-avainsana täsmentää testauksen kohteen, minkä jälkeen käytetään context- ja it-avainsanoja, jotka pyrkivät välittämään lukijalle tiedon siitä, mitä yksikkötesti käytännössä testaa.

```
class SearchResultTest: QuickSpec {

  override func spec() {
    describe("SearchResult") {
      context("should") {
        it("contain region as regionAndLocality") {
          let result = SearchResult.create(fromLocality: "")
          expect(result.region).to(equal(result.regionAndLocality))
        }

        it("contain locality as regionAndLocality") {
          let result = SearchResult.create(fromRegion: "")
          expect(result.locality).to(equal(result.regionAndLocality))
        }

        it("contain region and locality as regionAndLocality") {
          let result = SearchResult.create()
          let expected = result.locality + ", " + result.region
          expect(result.regionAndLocality).to(equal(expected))
        }
      }
    }
  }
}
```

Esimerkkikoodi 4. Yksikkötesti, jossa on käytetty Nimble- ja Quick-kirjastoja.

Esimerkkikoodista käy ilmi että erillistä test-prefiksiä ei tarvita kuten XCTest-kirjaston yksikkötesteissä ja kyseisen luokan kaikki testitapaukset on määritetty spec-metodin sisällä. Spec-metodi sisältää edellä mainitut avainsanat, joita ylhäältä alaspäin lukemalla saadaan selkeä testin kuvaus. Esimerkiksi esimerkkikoodin 4 kolmas testitapaus näkyy Xcode-editorissa muodossa ”SearchResult should contain and locality as regionAndLocality” eikä yhteen kirjoitettuna kuten XCTest-kirjastolla kirjoitetut testit.

Automatisoidut käyttöliittymätestit toteutettiin XCTest-kirjaston sisältämällä XCUITest-luokalla, jonka avulla sovellus käynnistetään erillisessä prosessissa ja sitä voidaan testata käyttäjän näkökulmasta. Käyttöliittymäkomponentit voidaan hakea XCUIApplication-luokan avulla, joka sisältää tarvittavat metodit esillä olevien komponenttien paikantamiseen esimerkiksi indeksin, tekstin tai komponentin luonteen perusteella.

Esimerkkikoodi 5 sisältää yhden käyttöliittymätestin, jonka tarkoituksena on varmistaa, että käyttöliittymästä on lähtöpiste ja määränpää valittuna, ennen kuin käyttäjä voi hakea reittitietoja.

```
class RouteSearchUITests: XCTestCase {

  var app: XCUIApplication!

  func testShouldEnableSearchWhenOriginAndDestinationHaveBeenSelected() {
    app = XCUIApplication()
    app.launch()

    let origin = app.textFields.firstMatch
    XCTAssertTrue(origin.isEnabled)

    let searchButton = app.buttons["Hae"]
    XCTAssertTrue(searchButton.exists)
    XCTAssertFalse(searchButton.isEnabled)

    // verify valitse is disabled when cell has not been selected
    let originText = "Kirkkonummi 1"
    selectLocation(origin, selectCellWithString: originText)
    XCTAssertTrue(app.textFields[originText].exists)

    // navigate to selection (select destination)
    let destination = app.textFields.allElementsBoundByIndex[1]
    XCTAssertTrue(destination.isEnabled)

    // verify valitse is disabled when cell has not been selected
    let destinationText = "Herttoniemi"
    selectLocation(destination, selectCellWithString: destinationText)
    XCTAssertTrue(app.textFields[destinationText].exists)

    XCTAssertTrue(searchButton.isEnabled)
  }
}
```

Esimerkkikoodi 5. Käyttöliittymätesti, jossa käytetään XCTest-kirjaston XCUITest-luokkaa.

4.5 Jatkuvan integraation käyttöönotto

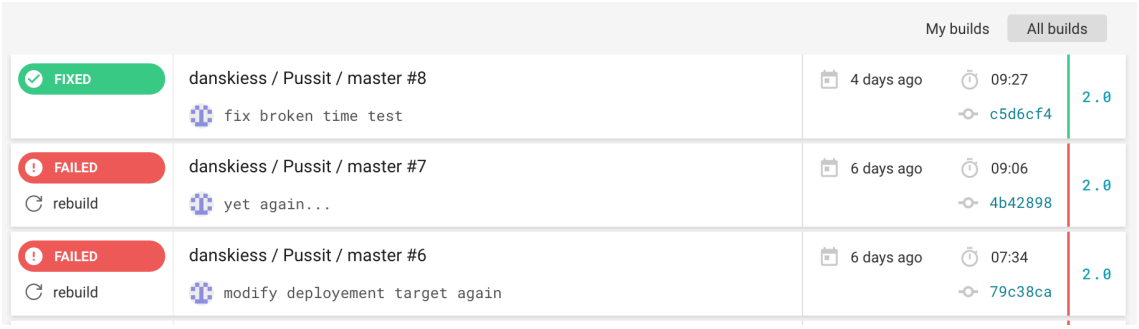
Jatkuva integraatio otettiin käyttöön sovelluksen kehityksen loppuvaiheessa, joka on varsin myöhäinen ajankohta eikä välttämättä kovinkaan hyödyllinen työkalu yksin työskennellessä. Joka tapauksessa se toteutettiin käyttämällä CircleCI-pilvipalvelua, joka on erittäin suoraviivainen tapa lisätä sovellukselle jatkuva integraatio, mutta haittapuolena on palvelun maksullisuus yksityisille projekteille. Kirjoitushetkellä se hakee sovelluksen lähdekoodin versionhallintajärjestelmästä muutoksen yhteydessä, minkä jälkeen se luo sovelluksesta koontiversion ja suorittaa automatisoidut testit. CircleCI antaa palautetta mikäli sovelluksen koontiversion luominen ei onnistu tai testejä ei suoritettu onnistuneesti.

Käyttöönotto edellytti konfiguraatiotiedoston luomisen (esimerkkikoodi 6), jossa määritetään automatisoidut työvaiheet, kuten lähdekoodin haku versionhallinnasta, sovelluksen koontiversion luonti ja automatisoitujen testien suoritus sekä halutut testauslaitteet, joilla sovelluksen testit suoritetaan.

```
version: 2
jobs:
  build:
    macos:
      xcode: "9.0"
    steps:
      - checkout
      - run:
          name: Install CocoaPods
          command: pod install
      - run:
          name: Build and run tests
          command: fastlane scan
          environment:
            SCAN_DEVICE: iPhone 6
            SCAN_SCHEME: Pussit
      - store_test_results:
          path: test_output/report.xml
      - store_artifacts:
          path: /tmp/test-results
          destination: scan-test-results
      - store_artifacts:
          path: ~/Library/Logs/scan
          destination: scan-logs
```

Esimerkkikoodi 6. Sovelluksen CircleCI-konfiguraatiotiedosto, config.yml.

Käyttöönoton jälkeen palvelun web-käyttöliittymää tarjoaa palautetta integraatioiden tilasta. Palautteessa (kuva 12) käy ilmi integraation tila ja sen suorittamiseen käytetty aika sekä versionhallinnasta saadut tiedot, kuten kehityshaara, tunniste ja muutosta kuvaava viesti.



Kuva 12. CircleCI-palvelun tarjoama palaute sovelluksen integraatioista

Kuvasta käy ilmi, että yhden integraation suorittaminen kestää lähes kymmenen minuuttia, joka on suhteellisen pitkä aika yksinkertaisen sovelluksen integraatiolle, vaikka käyttöliittymätestien suorittaminen on hitaampaa kuin yksikkötestien. Kirjoitushetkellä suurin osa integraatioon kuluvasta ajasta kuluu versionhallinnasta haettavan lähdekoodin ja sovelluksen riippuvuuksien lataamiseen sekä käyttöliittymätestien suorittamiseen. Halutessaan prosessia voi nopeuttaa tallentamalla ympäristön ja sen riippuvuudet, jotta ei tarvitse ladata niitä useita kertoja.

5 Yhteenveto

Mobiilisovellusten testaaminen on tärkeä osa mobiilisovelluskehitystä, koska käyttäjien vaatimukset ovat tavallista korkeammat ja käyttäjillä on matala kynnys poistaa toimimaton sovellus käytöstä. Loppukäyttäjien korkeat vaatimukset luovat tilanteen, jossa mobiilisovelluksien testaaminen on erittäin tärkeää eikä sovelluksen loppukäyttäjiä voi aliarvioida. Onnistuneen ja laadukkaan sovelluksen rakentaminen edellyttää tarjolla olevien työkalujen käyttöönottoa, koska selvitettävää ja testattavaa on huomattava määrä. Manuaalinen testaus saattaa joissakin tapauksissa olla aikaa säästävä ja tehokas tapa tehdä sovelluksen testausta, mutta sovelluksella tulee silti olla automatisoituja testejä.

Automatisoitujen testien avulla varmistutaan sovelluksen laadusta ja toimivuudesta, ja niiden avulla voidaan välttää sovelluksen kehityksessä tapahtuvat regressiot. Automatisoitujen testien luominen lähtee nykypäivänä jo kehittäjistä, koska he toimivat myös testaajien roolissa, mikä mahdollistaa laadun rakentamisen tuotteeseen jo sen kirjoitushetkellä. Ottamalla käyttöön vaihtoehtoiset testaustavat, kuten varhainen alfa- ja beeta-jakelu, saadaan paljon lisätietoa käyttäjiltä suoraan ja epäsuoraan. Käyttäjät voivat antaa suoraa palautetta sovelluksen toiminnasta, mikä auttaa sen kehittämisessä. Epäsuora palaute voi olla virheraportoinnin tai analytiikan muodossa, ja sen avulla voidaan selvittää sovelluksen epäkohtia ja korjata niitä.

Insinöörityön tarkoituksena oli perehtyä iOS-sovelluskehitykseen ja sen testaukseen, missä mielestäni onnistuttiin melko hyvin. Insinöörityön lopputuotteena on iOS-sovellus, joka kykenee näyttämään Reittioppaan reittitietoja ja aktiiviset häiriötilanteet. Sen lisäksi sovellus sisältää automatisoituja yksikkö-, integraatio- ja käyttöliittymätestejä. Sovellukseen lisättiin työn loppuvaiheessa jatkuva integraatio oman mielenkiinnon vuoksi. Se suorittaa automatisoidut testit jokaisesta versionhallintaan viedystä muutoksesta ja antaa palautetta sovelluksen toiminnasta.

Jatkon kannalta sovellusta voisi kehittää tulevaisuudessa monipuolisemmaksi, parantaa sen käyttöliittymää ja lisätä automatisoitujen testien lukumäärää. Sovellusta varten voisi luoda end-to-end-testejä, joilla sovellusta testataan nimensä mukaisesti päästä päähän eikä sijaiskomponentteja käytettäisi todellisten komponenttien sijasta. End-to-end-testeillä saadaan suurempi varmuus sovelluksen toiminnasta eikä verkkoyhteyksien testaamista tarvitse tehdä käsin. Sen lisäksi sovelluksen voisi lokalisoida eri kielille, joiden testausta varten voisi myös laatia automatisoituja testejä.

Lähteet

About Continuous Integration in Xcode. Verkkoaineisto. Apple. <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/xcode_guide-continuous_integration/>. Luettu 5.12.2017.

About Swift. Verkkoaineisto. Swift. <https://swift.org/about/#swiftorg-and-open-source/>. Luettu 26.10.2017.

Apple Developer Documentation. Verkkoaineisto. Apple. <https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html>. Luettu 29.10.2017.

Arsene, Codrin. 2016. 17 Strategies for End to End Mobile Testing on both iOS and Android. Verkkoaineisto. <https://ymedialabs.com/17-strategies-for-end-to-end-mobile-testing-on-both-ios-and-android/>. Luettu 17.10.2017.

Beta Testing Made Simple. Verkkoaineisto. Apple. <https://developer.apple.com/testflight/>. Luettu 5.12.2017.

What is Software Architecture? Verkkoaineisto. Microsoft. <https://msdn.microsoft.com/en-us/library/ee658098.aspx/>. Luettu 20.10.2017.

Architectural Patterns and Styles. Verkkoaineisto. Microsoft. <https://msdn.microsoft.com/en-us/library/ee658117.aspx/>. Luettu 23.10.2017.

Crispin, Lisa. 2009. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley Professional. Boston: Pearson Education Inc.

Cynthia Lee. 2014. Gartner Says Traditional Development Practices Will Fail for Mobile Apps. Verkkodokumentti. Gardner. <https://www.gartner.com/newsroom/id/2823619/>. Luettu 16.10.2017.

EarlGrey. Verkkodokumentti. Github. <https://github.com/google/EarlGrey/>. Luettu 26.10.2017.

Francino, Yvette. Master interrupt testing on mobile devices. Verkkoaineisto. < http://searchsoftwarequality.techtarget.com/tip/Master-interrupt-testing-on-mobile-devices>. Luettu 17.10.2017.

Gartner Says Traditional Development Practices Will Fail for Mobile Apps. 2014. Verkkodokumentti. Gartner. <https://www.gartner.com/newsroom/id/2823619/>. Luettu 16.10.2017.

Glezos, Dimitris. Localization 101: A Beginner's Guide to Software Localization. Verkkoaineisto. <https://www.transifex.com/blog/2015/software-localization-guide/>. Luettu 17.10.2017.

Haikala, Mikko & Mikkonen, Tommi. 2011. Ohjelmistotuotannon käytännöt. Alma Talent.

Hechtel Ely. 2016a. What are Mobile App Testing Challenges? Verkkoaineisto. Saucelabs. < https://saucelabs.com/blog/what-are-mobile-app-testing-challenges/>. Luettu 16.10.2017.

Hechtel Ely. 2016b. Mobile Device Emulator vs Simulator vs Real Device. Verkkoaineisto. Saucelabs. <https://saucelabs.com/blog/mobile-device-emulator-and-simulator-vs-real-device/>. Luettu 8.12.2017.

Helppi, Ville-Veikko. 2016a. How to Get Started with EarlGrey – iOS Functional UI Testing Framework. Verkkoaineisto. Bitbar. <https://bitbar.com/how-to-get-started-with-earlgrey-ios-functional-ui-testing-framework/>. Luettu 26.10.2017.

Helppi, Ville-Veikko. 2016b. Pros and Cons of Using XCTest for iOS Testing. Verkkoaineisto. Bitbar. <https://bitbar.com/pros-and-cons-of-using-xctest-for-ios-testing/>. Luettu 10.11.2017.

Janga, Siddartha. 2016. EarlGrey – iOS Functional UI Testing Framework. Verkkoaineisto. Google. <https://developers.googleblog.com/2016/02/earlgrey-ios-functional-ui-testing.html>. Luettu 26.10.2017.

Kaczanowski, Tomek. 2013. Practical Unit Testing with JUnit and Mockito. Kaczanowski, Tomasz. Verkkojulkaisu.

Kasurinen, Jukka Pekka. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.

Knott, Daniel. 2015. Hands-On Mobile App Testing: A Guide for Mobile Testers and Anyone Involved in the Mobile App Business. Indiana: Addison-Wesley Professional.

Kolodiy, Sergey. Unit Tests, How to Write Testable Code and Why it Matters. Verkkoaineisto. Toptal. <https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters>. Luettu 11.03.2017.

Laboon, Bill. 2016. A Friendly Introduction to Software Testing. CreateSpace Independent Publishing Platform. Verkkojulkaisu.

Lazinskiy, Lev. 2017. What is Continuous Integration? Verkkoaineisto. CircleCI. <https://circleci.com/blog/what-is-continuous-integration/>. Luettu 4.12.2017.

Manferdini, Matteo. 2016. A Better iOS Architecture: A Deep Look At The Model-View-Controller Pattern. Verkkoaineisto. Smashing magazine. <https://www.smashingmagazine.com/2016/05/better-architecture-for-ios-apps-model-view-controller-pattern/>. Luettu 20.10.2017.

Mobile Testing: Manual Vs. Automation. 2016. Verkkoaineisto. Orasi. <https://www.orasi.com/wp-content/uploads/2016/12/Manual_Vs_Automation_Mobile_Testing_WP.pdf>. Luettu 8.12.2017.

Model-View-Controller. Verkkoaineisto. Apple. <https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>. Luettu 20.10.2017.

Myers, Glenford J. 2011. The Art of Software Testing. New Jersey: John Wiley & Songs Inc.

Nelson, Tom. 2017. macOS: What Is It and What's New? Verkkoaineisto. <https://www.lifewire.com/what-is-macos-4144656>. Luettu 26.10.2017.

Nimble. Verkkoaineisto. Github. <https://github.com/Quick/Nimble/>. Luettu 28.10.2017.

Quick and Nimble Testing with Swift. Verkkoaineisto. IBM. <https://www.ibm.com/innovation/milab/quick-and-nimble-testing-with-swift/>. Luettu 28.10.2017.

Santarossa, Marcus. 2017. Verkkoaineisto. <https://marcosantadev.com/mvvmc-with-swift/>. Luettu 17.10.2017.

Sharkley, Kent. 2016. Localization Testing. Verkkoaineisto. <https://docs.microsoft.com/en-us/globalization/testing/localization-testing/>. Luettu 17.10.2017.

Signing workflow. Verkkoaineisto. Apple. <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppStoreDistributionTutorial/Setup/Setup.html/>. Luettu 26.10.2017.

St. Pierre, Jason. 2014. Launching Beta by Crashlytics. Verkkoaineisto. Fabric. <https://fabric.io/blog/2014/05/29/launching-beta-by-crashlytics/>. Luettu 6.12.2017.

Tarlinder, Alexander. 2016. Developer Testing: Building Quality into Software. Addison-Wesley Professional.

TestFlight beta testing overview (iOS, tvOS, watchOS). Verkkoaineisto. Apple. <https://help.apple.com/itunes-connect/developer/#/devdc42b26b8/>. Luettu 5.12.2017.

Testing Apps with TestFlight. Verkkoaineisto. Apple. <https://developer.apple.com/testflight/testers/>. Luettu 5.12.2017.

Testing Basics. Verkkoaineisto. Apple.
<https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/03-testing_basics.html#//apple_ref/doc/uid/TP40014132->. Luettu 10.11.2017.

Tools you'll love to use. Verkkoaineisto. Apple.
<https://developer.apple.com/xcode/ide/>. Luettu 26.10.2017.

Wackler, Mike. 2015. Just Say No to More End-to-End Tests. Verkkoaineisto. Google.
<https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html>.
Luettu 10.3.2017.

XCTest Documentation. Verkkoaineisto. Apple.
<https://developer.apple.com/documentation/xctest>. Luettu 10.11.2017.

Zablocki, Krzysztof. 2017. Testing iOS Apps. Verkkoaineisto.
<http://merowing.info/2017/01/testing-ios-apps/>. Luettu 6.11.2017.